NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA

THESIS

ANALYZING THE INTEL PENTIUM'S CAPABILITY TO SUPPORT A SECURE VIRTUAL MACHINE MONITOR

by

John Scott Robin

September 1999

Thesis Advisor: Cynthia Irvine
Second Reader: Steven B. Lipner

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE September 1999	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE ANALYZING THE INTEL PENTIUM'S CAPABILITY TO SUPPORT A SECURE VIRTUAL MACHINE MONITOR

5. FUNDING NUMBERS

6. AUTHOR(S) Robin, John S.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

This thesis addresses the problem of implementing secure virtual machine monitors (VMM) on the Intel Pentium architecture. A VMM allows multiple operating systems to run concurrently under virtual machines on a single workstation. High-assurance VMMs could allow complete isolation of, or data sharing between, virtual machines according to a security policy such as a mandatory secrecy policy.

The Intel architecture was mapped to a set of hardware requirements for VMMs. It was found that the Intel architecture was not virtualizable. However, several techniques are presented that allow the Intel architecture to support a "virtual VMM." A commercial virtual VMM was studied and found to be unable to support secure VMMs. Therefore, a foundation upon which a secure VMM could be built for the Intel Pentium architecture is presented.

A secure VMM for the Intel architecture offers several benefits. First, PC users could work in a more secure environment. Second, PC users could run familiar COTS operating systems and applications. Finally, secure VMMs could save the DoD millions of dollars by eliminating the need for separate systems when both high assurance, and COTS operating systems and applications are required.

14. SUBJECT TERMS Virtual Machines, Virtual Machine Monitors, Intel Architecture, Multilevel Security, Intel Pentium			15. NUMBER OF PAGES 113
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18 298-102

Approved for public release; distribution is unlimited.

ANALYZING THE INTEL PENTIUM'S CAPABILITY TO SUPPORT A SECURE VIRTUAL MACHINE MONITOR

John Scott Robin
Second Lieutenant, United States Air Force
B.S., United States Air Force Academy, 1998

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 1999

Author: John Scott Robin
John Scott Robin

Approved by: Cynthia E. Irvine
Cynthia Irvine, Thesis Advisor

Steven B. Lipner, Thesis Second Reader

Dan C Boger
Dan Boger, Chairman
Department of Electrical and Computer Engineering

ABSTRACT

This thesis addresses the problem of implementing secure virtual machine monitors (VMM) on the Intel Pentium architecture. A VMM allows multiple operating systems to run concurrently under virtual machines on a single workstation. High-assurance VMMs could allow complete isolation of, or data sharing between, virtual machines according to a security policy such as a mandatory secrecy policy.

The Intel architecture was mapped to a set of hardware requirements for VMMs. It was found that the Intel architecture was not virtualizable. However, several techniques are presented that allow the Intel architecture to support a "virtual VMM." A commercial virtual VMM was studied and found to be unable to support secure VMMs. Therefore, a foundation upon which a secure VMM could be built for the Intel Pentium architecture is presented.

A secure VMM for the Intel architecture offers several benefits. First, PC users could work in a more secure environment. Second, PC users could run familiar COTS operating systems and applications. Finally, secure VMMs could save the DoD millions of dollars by eliminating the need for separate systems when both high assurance, and COTS operating systems and applications are required.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

I. INTRODUCTION

A. BACKGROUND

Technological developments in the early 1970's brought about large multi-access, multi-programming, multi-processing computer systems. Multi-access allowed many users to access the same computer simultaneously. Multi-programming allowed multiple programs to be loaded into a computer's memory simultaneously. A scheduler in the system time-multiplexes the processor among the processes that are executing the programs. Finally, multi-processing allowed one computer to use many processors simultaneously.

The characteristics listed above brought about a new age in computing. Most computer users no longer had to be in the computing facility to use the computer. However, this was not true for system programmers. System programmers require direct access to the resources of the computer system, which can not be provided through the computer's operating system. Therefore, not only did system programmers have to be in the computing facility to work, they had to make the system inaccessible to other users in order to do their work. To overcome this difficulty, virtual machine monitors were invented. A virtual machine monitor provides all computer system users the appearance of having direct access to the resources of a "bare" computer.

A virtual machine monitor (VMM) is software for a computer system that creates efficient, isolated programming environments that are "duplicates" of the real machine environment. These "duplicates" are referred to as virtual machines. Goldberg defines a virtual machine (VM) as: "a hardware-software duplicate of a real existing computer system in which a statistically dominant subset of the virtual processor's instructions execute on the host processor in native mode" [Ref. 1]. A VMM mediates between the virtual machine(s) and the real resources of the computer system. CP-67 is an example of one of the earliest virtual machine monitors.

In the past, some virtual machine monitors have been used to separate mandatory security classes. Examples of such usage can be seen in SDC's KVM-370 and the DEC VAX SVS. If a secure VMM could be built for the Intel Pentium[1] architecture, it would be very attractive because a single machine could be used to implement multi-level security and also run commercial-off-the-shelf operating systems and applications. The focus of this research is to determine whether a VMM written for the Intel Pentium architecture can provide this type of security and convenience.

B. GOALS OF THE THESIS

The first goal of this thesis is to determine whether any type of virtual machine monitor can be built on the Intel Pentium architecture. Types of virtual machine monitors include Type I, Type II, and Hybrid (all of which are described later in this thesis). If a VMM can be constructed for the Intel Pentium architecture, I will determine whether the VMM can be secure.

C. VIRTUAL MACHINE MONITORS

1. Benefits of Virtual Machine Monitors

Virtual machine monitors have many benefits. First, virtual machine monitors normally allow a system manager to configure the environment in which a virtual machine will run. Therefore, virtual machines can have configurations different from those of the real machine. This means that even though a real machine might have 32MB of memory, a virtual machine may be set to have 8 MB of memory. This situation would allow a developer to test the performance of his application on a machine with only 8 MB of memory without having to construct a real machine with only 8 MB of memory.

Second, virtual machines allow many different operating systems to be run

---

[1] Throughout this paper, the term "Intel Pentium architecture" will refer to the architecture of the following processors which are all trademarks of the Intel Corporation: Intel Pentium, Intel Pentium Pro, Intel Pentium with MMX Technology, Intel Pentium II, and Intel Pentium III.

concurrently on the same computer. Users can run any operating system and applications they choose as long as they are designed to run on the real processor architecture. This benefit makes application development for different operating systems much easier. Since several different operating systems can run on the same computer, a developer can test his application on many operating systems using the same computer.

Third, virtual machines allow users to run untrusted applications in an isolated environment. For example, a program that is downloaded from the Internet could be tested in a virtual machine. If the program contained a virus, the virus would be isolated to that virtual machine. This protects the rest of the machine's applications and data.

Fourth, virtual machines can be used to upgrade operating system software to a different version without losing the ability to run the older "legacy" operating system and its applications. The legacy operating system and its applications can run in a virtual machine exactly as they did previously on the real machine, while the new version of the operating system runs in a separate virtual machine.

Finally, virtual machine monitors can be used to construct system software for scalable computers that have anywhere from 10 to 100 processors. These systems are being used more and more in the marketplace. However, the system software for these scalable machines has not reached the functionality and reliability that is expected in modern operating systems. Operating system developers must be blamed for this problem. They must make many modifications to an operating system to support scalable machines.

Virtual machine monitors are a solution to this problem. Using a VMM, an additional software layer can be inserted between the hardware and multiple operating systems. This layer would allow multiple copies of an operating system to run on the same scalable computer. The VMM also allows these operating systems to share resources with each other. This solution has most of the features of an operating system that was custom-built for a scalable machine. However, the development costs and complexity of the virtual machine monitor are significantly lower than for a custom solution. A prototype of this solution, called Disco, was developed at Stanford University

on the Stanford FLASH shared-memory multi-processor [Ref. 2]. Disco uses many different commercial operating systems to provide high-performance system software. The professors at Stanford who worked on the Disco prototype later formed VMware, Inc. Their product, called VMware[2], is a VMM for the Intel Pentium architecture and is discussed in a portion of this thesis.

2. Characteristics and Layers of a VMM

A VMM has three characteristics [Ref. 3]. First, a VMM provides an environment that is almost identical to the original machine. This means that any program that runs in a VM should run the same as if it had been run on the original machine. The exceptions to this rule are differences in system resource availability, timing dependencies, and attached I/O devices. If resource availability is different, such as reduced physical memory, the program will obviously not perform as well because the program will need to page or swap. Timing dependencies may lose their validity because a VMM may intervene and execute a different set of instructions when certain instructions are executed by a VM. These substitute instructions may take longer than expected to execute. Therefore, any assumptions about how long instructions will take to execute may be incorrect. Finally, if the VM is not configured to have a peripheral device that is attached to the real machine, such as a network card, it will not be able to access the peripheral device even though it is attached to the real machine.

The second characteristic of a VMM is that it must be in control of real system resources. This means that no program running under a VMM can access any resource that is not explicitly allocated to it by the VMM. It also means that it is possible for the VMM to regain control of resources that it already allocated.

Third, a VMM must be efficient. This means that a large percentage of the virtual processor's instructions must be executed by the machine's real processor, without VMM intervention. Instructions which can not be executed directly by the real processor are interpreted by the VMM.

---

[2] VMware and VMware's patent-pending Virtual Platform are trademarks of VMware, Inc.

Some virtual machines exhibit the recursion property. This means that it is possible to run a VMM inside of a VM, producing a new level of virtual machines. The real machine is normally called Level 0. A VMM running on Level 0 is said to be Level 1, etc.

3. Logical VMM Modules

A VMM normally has three generic types of modules: dispatcher, allocator, and interpreter. A jump to the dispatcher is placed in every location to which the machine traps. The dispatcher then decides which of its modules to call when the machine traps. The second type of module is the allocator. If a VM tries to execute a privileged instruction that would change the resources of the VM's environment, the VM will trap to the VMM dispatcher. The dispatcher will handle the trap by invoking the allocator that performs the requested resource allocation according to VMM policy. There is only one allocator module in a VMM. However, the allocator is a large portion of the virtual machine monitor. It decides which system resources to provide to each VM, ensuring that two different VM's do not get the same resource. The final module type is an interpreter. Each privileged instruction will have an interpreter module that is called by the dispatcher to simulate the effect of the instruction that caused the trap.

D. THESIS ORGANIZATION

The rest of this thesis is organized as follows: Chapter II contains a discussion of the three different types of VMMs and their hardware requirements. Chapter III is an analysis of the Intel architecture to determine if it can meet any of the VMM hardware requirements described in Chapter II. Chapter IV is a case study of a commercial product called VMware and how it relates to Intel Pentium virtualization. Chapter V determines whether or not a VMM designed for the Intel Pentium architecture can be secure. Finally, Chapter VI is a conclusion of this work and also addresses possible future research.

This thesis has two appendices. Appendix A contains a brief summary of the Intel architecture and is recommended to readers who are not familiar with the Intel Pentium architecture. Appendix B is a table of the results obtained from the analysis in Chapter

III. It is a list of all documented Intel Pentium instructions and whether or not they are virtualizable.

II. TYPES OF VIRTUAL MACHINE MONITORS

This chapter discusses each type of VMM including the Type I VMM, Type II VMM, and Hybrid VMM. It will also cover the architectural features that each type of VMM requires in order to be implemented.

An operating system consists of instructions to be executed on a hardware processor. When an operating system is virtualized, some portion, ranging from none to all, of the instructions may be executed by underlying software. The amount of software and hardware execution of processor instructions determines if one has a complete software interpreter machine (CSIM), hybrid VM (HVM), VMM, or a real machine. Each of these different types of machines provides a normal machine environment, meaning that processor instructions can be executed in them. Thus, a VMM can host an operating system. However, they differ in the way that the machine environment actually executes the processor instructions. A real machine uses only direct execution, meaning that the processor executes every instruction. A CSIM uses only software interpretation, meaning that a software program emulates every processor instruction. Goldberg says that a VMM requires that a "statistically dominant subset" of the virtual processor's instructions be executed on the real processor [Ref. 1]. Although he does not give a specific percentage, it is easy to see that performance will be better if more instructions are executed directly by the processor. For example, when executed directly, the LGDT *x* instruction loads *x* into the global descriptor table register of the processor. However, when emulated by software, the instruction would first trap to a VMM. The VMM would then execute a MOV *y*, *x* instruction to store the value *x* in a special memory location *y* designated by the VMM. The special memory location, *y*, is necessary because the processor's global descriptor table register (GDTR) must hold the location of the global descriptor table of the VMM or host OS. A VMM can not allow a virtual machine to overwrite the real GDTR register. Finally, the VMM would have to return to the virtual machine to allow it to continue executing. VMMs primarily use direct execution, with

occasional traps to software. As a result, the performance of VMMs is better than CSIMs and HVMs. An HVM is a VMM that uses software interpretation on all privileged instructions. HVMs are possible on a larger class of systems than VMMs.

The definition of a VMM does not specify how the VMM gains control of the machine to interpret instructions that cannot be directly executed on the processor. As a result, there are two different types of VMMs that can create a virtual machine environment. These types are referred as Type I and Type II in Goldberg's thesis. A Type I VMM runs on a bare machine, meaning that it is an operating system with virtualization mechanisms. It performs the scheduling and allocation of the system's resources. A Type II VMM runs as an application under an operating system. The operating system that controls the real hardware of the machine is called the "host OS." The host OS does not need or use any part of the virtualization environment. Every OS that is run in the virtual environment under the host OS is called a "guest OS." In a Type II VMM, the host operating system provides resource allocation and a standard execution environment to each guest OS.

When executing in a virtual machine, some processor instructions can not be executed directly on the processor. These instructions would interfere with the state of the underlying VMM or host OS and are called sensitive instructions. The key to implementing a VMM is to prevent the direct execution of sensitive instructions. Some sensitive instructions in the Intel Pentium architecture are privileged, meaning that if they are not executed at CPL 0, they will cause a general protection exception. Normally, a VMM is executed in privileged mode and a VM is run in user mode. When sensitive instructions that are privileged are executed in a VM, they cause a trap to the VMM. If all sensitive instructions of a processor are privileged, the processor is considered to be "virtualizable." This is because all sensitive, privileged instructions will trap to the VMM because they are executing in user mode. After trapping, the VMM will execute code that will emulate the proper behavior of the privileged instruction for the virtual machine. However, if sensitive, non-privileged instructions exist, they may interfere

with the proper operation of the VM. This means that it may be necessary for the VMM to examine each instruction before execution to ensure that it is not a sensitive, non-privileged instruction. When a sensitive instruction is encountered, the virtual machine must be forced to trap to the VMM so that it can be handled properly. Examining every instruction before it is executed will cause considerable overhead.

The most severe performance penalty comes when running a complete software interpreter machine (CSIM). A CSIM emulates every instruction of the real processor. It is not a virtual machine because it does not execute any of the instructions directly on the real processor. Figure 1 below illustrates the various types of machines based on the amount of hardware and software execution.

In his thesis, Goldberg explores third generation hardware to determine which processors, if any, can run virtual machine monitors. Some of the key architectural features of third generation hardware are: two processor modes of operation, a method for non-privileged programs to call privileged system routines, a memory relocation or protection mechanism such as segmentation or paging, and asynchronous interrupts to allow the I/O system to communicate with CPU. Even though Goldberg's thesis was written over twenty-five years ago, all of these characteristics still apply to the Intel Pentium architecture. The Intel Pentium processor has four modes of operation, known as rings, or current privilege level (CPL), 0 through 3. Ring 0 is the most privileged level of operation. Operating systems operate in this ring. Ring 3 is the least privileged ring, where applications programs execute. The Intel architecture also has a method for non-privileged tasks to call privileged system routines—the call gate. Call gates allow transfer of program control between privilege levels. The Intel architecture also uses both paging and segmentation to implement its protection mechanism. Finally, the Intel architecture uses both interrupts and exceptions to allow the I/O system to communicate with the CPU. The architecture has 16 predefined interrupts and exceptions and 224 user-defined, or maskable, interrupts.

After examining third generation hardware, Goldberg developed a list of requirements that a processor must meet in order to be virtualizable. He analyzed twelve different third generation processors and found that only five were virtualizable.

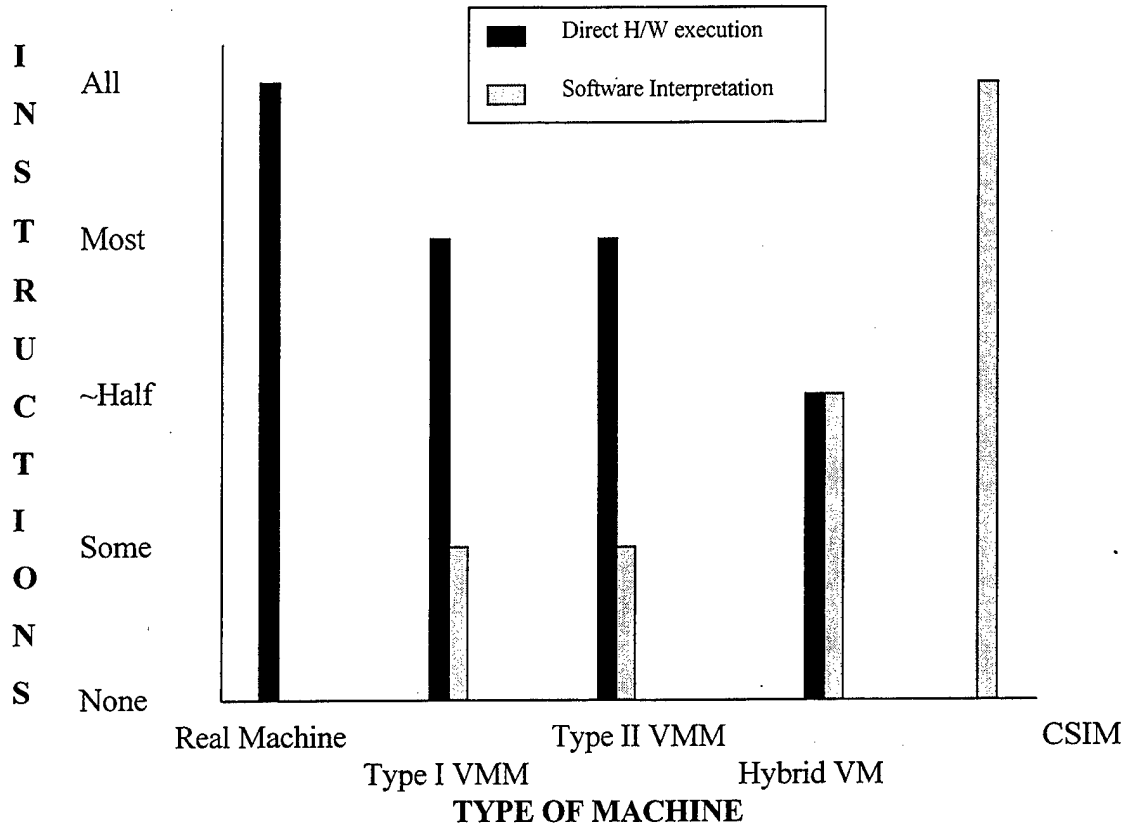


Figure 1. Hardware and Software Execution of Various Types of Machines.

He concluded that most of the seven remaining processors were not virtualizable because they were not designed to be. The following sections will analyze each type of VMM and the requirements that a processor must meet in order to support it.

A. TYPE I VMM

A Type I VMM runs directly on the machine hardware. In other words, it is an operating system or kernel that has mechanisms to support virtual machines. A Type I VMM is illustrated in Figure 2.

A Type I VMM must perform scheduling and resource allocation for all virtual machines in the system. This means that a Type I VMM may be much larger than Type II VMM because of the extra code needed to implement these features. Furthermore, a Type I VMM requires drivers for hardware peripherals.

Goldberg develops a set of rules to determine if processor hardware is capable of supporting virtual machines and thus could be a host for a Type I VMM. His three requirements for virtualization are:

1) The method of executing non-privileged instructions must be roughly equivalent in both privileged and user mode. For example, a processor can not use an additional bit in an instruction word or in the address portion of an instruction when in privileged mode.

2) There must be a method such as a protection system or an address translation system to protect the real system and any other VMs from the active VM.

VM 1	VM 2	VM 3	VM 4
Apps	Apps	Apps	Apps
Windows 98	Windows NT	Other Intel OS	Linux
Type I VMM			
Hardware			

Figure 2. A Hypothetical Type I VMM Supporting Popular PC Operating Systems.

3) There must be a way to automatically signal the VMM when a VM attempts to execute a sensitive instruction. It must also be possible for the VMM to simulate the effect of the instruction. Sensitive instructions include:

A) Instructions that attempt to change or reference the mode of the VM or the state of the machine.

B) Instructions that read or change sensitive registers and/or memory locations such as a clock register and interrupt registers.

C) Instructions that reference the storage protection system, memory system, or address relocation system. This class includes instructions that would allow the VM to access any location that is not in its virtual memory.

D) All I/O instructions.

B. TYPE II VMM

A Type II VMM runs as an application under a host operating system. A type II VMM is illustrated in Figure 3.

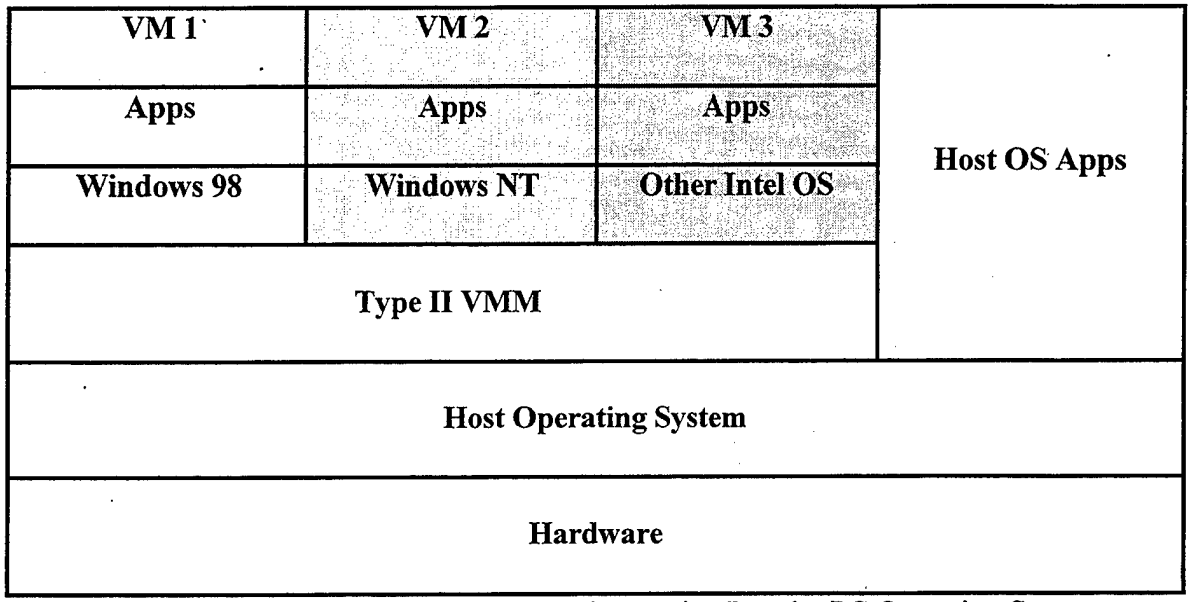


Figure 3. A Hypothetical Type II VMM Supporting Popular PC Operating Systems.

A Type II VMM should be simpler than a Type I VMM because the memory management, processor scheduling, resource allocation, and hardware drivers of the host operating system are used in its implementation. A Type II VMM provides only virtualization support services. The Type II VMM virtualizes the real machine even though the VMM is running as an application in the host OS.

To support a Type II virtual machine a processor must meet all of the hardware requirements for the Type I VMM listed above. However, in addition to these requirements, there are software requirements for the host operating system that a Type II VMM runs on. The host OS requirements are:

1) The host OS can not do anything to invalidate the requirement that the method of executing non-privileged instructions must be roughly equivalent in both privileged and user mode.

2) There must be primitives available in the host OS to protect the VMM and other VMs from the active virtual machine. Examples of this primitive include a protection primitive, address translation primitive, or a sub-process primitive.

When the virtual machine traps because it attempted to execute a sensitive instruction, the host OS must direct the signal to the VMM. Therefore, the host OS needs a primitive to perform this action. The host OS also needs a mechanism to allow a VMM to run the virtual machine as a sub-process. The VMM must still be able to simulate sensitive instructions.

This thesis will not analyze software requirements for a host OS because we are interested in analyzing the Intel processor's capability to run a secure VMM. A VMM that runs under an existing commercial-off-the-shelf OS that was designed for the Intel Pentium architecture such as Windows, Linux, and FreeBSD will in all likelihood not have a high level of security (it should be noted, however, that high assurance operating systems have been developed for the Intel architecture such as the WANG XTS 300 STOP [Ref. 4] and the Gemini GTNP GEMSOS [Ref. 5]). Therefore, a secure implementation of a VMM will almost certainly have to be a Type I VMM.

As with third generation processors, Goldberg found that most host operating systems (4 out of the 5 that were examined) could not support virtual machines. Again, this is because the operating systems were not designed with virtualization in mind.

C. HYBRID VMM

Often, if a processor does not meet the Type I or Type II VMM requirements, it can still implement a hybrid virtual machine monitor. A hybrid VMM has all of the advantages of normal VMMs and avoids the performance penalties of a CSIM. A hybrid virtual machine is functionally equivalent to the real machine. The major difference between an HVM and a VMM is that an HVM interprets every privileged instruction in software, whereas a VMM may directly execute some privileged instructions. Therefore, it treats the privileged mode of hardware as a pure software construct. In both a VMM and an HVM, all non-privileged instructions execute directly on the processor.

An HVM has less strict hardware requirements than a VMM for two reasons. First, the HVM does not have to directly execute non-sensitive privileged instructions because they are all emulated in software. Second, because of the emulation, the HVM does not have to map the most privileged processor mode into another privilege level of the processor. Performance of an HVM is usually lower than that of a VMM as a result of a higher number of privileged instructions being interpreted instead of being executed directly on the hardware.

The hardware requirements for an HVM to be virtualizable are changed in the following ways. First, requirement 1, which states that the method of executing non-privileged instructions must be roughly equivalent in both privileged and user mode, is eliminated. Second, requirement 3A, which states that if an instruction attempts to change or reference the mode of the VM or the state of the machine, there must be a way to automatically signal the VMM and a way to simulate the instruction, is weakened. Of the seven third generation processors that failed Goldberg's analysis for VMM, four of them were candidates for an HVM because of the less strict requirements.

Now that each type of VMM and its hardware requirements have been defined, it is necessary to examine the Intel Pentium architecture to see if it can support any of these VMMs.

THIS PAGE INTENTIONALLY LEFT BLANK

III. CAN THE INTEL ARCHITECTURE SUPPORT A VIRTUAL MACHINE MONITOR?

This chapter will analyze whether or not the Intel Pentium architecture is virtualizable by using the hardware requirements that were described in the previous chapter. A major result of this analysis is Appendix B, which contains every documented instruction for the Intel Pentium architecture and whether or not it supports virtualization.

Whether or not the Intel architecture is virtualizable is "a hit-or-miss proposition" because it was not designed to support virtual machines. After examining the processor requirements for virtualization, it can be seen that any instruction in the processor's instruction set that violates rule 1, 2, 3A, 3B, 3C, or 3D prevents the processor from running a Type I or Type II VMM. Additionally, any instruction that violates rule 2, 3A in its weaker form, 3B, 3C, or 3D prevents the processor from running an HVM. By combining these two statements, one can see that any instruction that violates rule 2, 3A in its weaker form, 3B, 3C, or 3D makes the processor non-virtualizable.

With respect to the VMM hardware requirements listed above, Intel meets all three of the main requirements for virtualization.

Requirement 1: The method of executing non-privileged instructions must be roughly equivalent in both privileged and user mode. Intel meets this requirement because the method for executing privileged and non-privileged instructions is the same. The only difference between the two types of instructions in the Intel architecture is that privileged instructions cause a general protection exception if the CPL is not equal to 0.

Requirement 2: There must be a method such as a protection system or an address translation system to protect the real system and any other VMs from the active VM. Intel uses both segmentation and paging to implement its protection mechanism. Paging is a mechanism where sections of a program's execution environment are mapped into physical memory as they are needed. Segmentation provides a mechanism to divide the linear address space into individually protected address spaces (segments). Segments are

used to hold the code, data, and stack for programs and to hold system data structures such as task state segment and a local descriptor table. Segments have a descriptor privilege level (DPL) ranging from zero to three that specifies the privilege level of the segment. The DPL is used to control access to the segment. Using DPLs, the processor enforces the boundaries between segments and does not allow one program to write into another program's segments.

Requirement 3: There must be a way to automatically signal the VMM when a VM attempts to execute a sensitive instruction. It must also be possible for the VMM to simulate the effect of the instruction. The Intel architecture uses interrupts and exceptions to redirect program execution and allow interrupt and exception handlers to execute when a privileged instruction is executed by an unprivileged task. However, the Intel instruction set contains instructions that are sensitive and unprivileged. The processor will execute unprivileged, sensitive instructions without generating an interrupt or exception. Therefore, a VMM will never have the opportunity to simulate the effect of the instruction.

After examining each member of the Intel instruction set (as of 20 June 99), it was found that seventeen instructions violate requirement 3. All seventeen instructions violate either part B or part C of requirement 3. An analysis of every Intel instruction can be found in Appendix C. The list in Appendix C contains the instruction name, its class, whether it is sensitive, whether it is privileged, whether it prevents virtualization, and the reason (if any) why it prevents virtualization. Each of the instructions that make the Intel processor non-virtualizable will be discussed in more detail below. Any manufacturer who wishes to make a version of the Intel Pentium chip that is truly virtualizable would need to focus on these instructions. Since all seventeen of the instructions to be discussed violate one of two requirements, there is a considerable amount of overlap in the discussion of each of instructions.

A. INSTRUCTIONS THAT REFERENCE OR CHANGE SENSITIVE REGISTERS (RULE 3B)

Several Intel instructions break hardware virtualization rule 3B. The rule states that instructions are sensitive if they read or change sensitive registers and/or memory locations such as a clock register and interrupt registers.

1. SGDT, SIDT, and SLDT Instructions

The SGDT, SIDT, and SLDT instructions are similar in the way that they violate this rule. In protected mode, all memory accesses pass through either the GDT or LDT. The GDT and LDT contain segment descriptors that provide the base address, access rights, type, length, and usage information for each segment. The interrupt descriptor table (IDT) is similar to the GDT and LDT, but it holds gate descriptors that provide access to interrupt and exception handlers. The GDTR, LDTR, and IDTR all contain the linear addresses and sizes of their respective tables.

All three of these instructions (SGDT, SIDT, SLDT) store a special register value into some location. The SGDT instruction stores the contents of the GDTR in a 6-byte memory location. The SLDT instruction stores the segment selector from the LDTR in a 16 or 32-bit general-purpose register or memory location. The SIDT instruction stores the contents of the IDTR in a 6-byte memory location. These instructions are normally only used by operating systems but are not privileged in the Intel architecture. Since the Intel processor only has one LDTR, IDTR, and GDTR, a problem arises when multiple operating systems try to use the same registers. Even though these instructions do not protect the sensitive registers from reading by unprivileged software, the Intel processor allows partial protection for these registers by only allowing tasks at CPL 0 to load the registers. This means that if a VM tries to write to one of these registers, a trap will be generated. The trap allows a VMM to produce the expected result for the VM. However, if an OS in a VM uses SGDT, SLDT, or SIDT to reference the contents of the IDTR, LDTR, or GDTR, the register contents that are applicable to the host OS or Type I VMM will be given. This could cause a problem if an operating system of a virtual machine (VMOS) tries to use these values for its own operations since they are what the VMOS

expects. Therefore, a Type I VMM or Type II VMM must provide each VM with its own virtual set of IDTR, LDTR, and GDTR registers.

2. SMSW Instruction

The SMSW instruction stores the machine status word (bits 0 through 15 of control register 0) into a general-purpose register or memory location. Bits 6 through 15 of CR0 are reserved bits that are not supposed to be modified. Bits 0 through 5, however, contain system flags that control the operating mode and state of the processor. These six bits are described in Table 1:

Bit Number	Flag Name	Description
0	PE – Protection Enable	Enable protected mode when set and real-mode when clear.
1	MP – Monitor Coprocessor	Controls the interaction of the WAIT or FWAIT instruction with the TS flag.
2	EM – Emulation	Indicates that the processor has an internal or external floating point unit when clear.
3	TS – Task Switched	Allows delayed saving of the floating point unit context on a task switch until the unit is accessed by the new task.
4	ET – Extension Type	For 386 and 468 processors, indicates whether an Intel 387 DX math coprocessor is present (hardcoded to 1 on >Pentium processors).
5	NE - Numeric Error	Enables the internal mechanism or PC-style mechanism for FPU error reporting.

Table 1. Important CR0 Machine Status Word Bits.

Even though this instruction only stores the machine status word, it is still sensitive and unprivileged. Consider the following scenario: A VMOS is running in real mode under the virtual environment of a VMM running in protected mode on the processor. If the VMOS checked the MSW to see if it was in real mode, it would incorrectly see that the PE bit is set. This means that the machine is in protected mode. If the VMOS halts or shuts down if in protected mode, the VMOS will not be able to run successfully.

According to the Intel instruction set reference [Ref. 6], this instruction is only in the architecture to be backward compatible with the Intel 286 processor. Programs written for the Intel 386 processor and later are supposed to use the MOV instruction to load and store control registers. Furthermore, the MOV to and from control register instructions are privileged instructions. Therefore, the SMSW could be removed from the Intel architecture completely and would only affect systems that needed to be backward compatible with the Intel 286 processor. Application software written for the Intel 286 and 8086 processors should be unaffected because the SMSW instruction is a system instruction that should not be used by application software.

3. PUSHF and POPF Instructions

The PUSHF and POPF instructions each reverse the operation of the other. The PUSHF instruction pushes the lower 16 bits of the EFLAGS register onto the stack and decrements the stack pointer by 2. The POPF instruction pops a word from the top of the stack, increments the stack pointer by 2, and stores the value in the lower 16 bits of the EFLAGS register. The PUSHFD and POPFD instructions are the 32-bit counter-parts of the POPF and PUSHF instructions. Pushing the EFLAGS register onto the stack allows the contents of the EFLAGS register to be examined. Much like the lower 16 bits of the CR0 register mentioned above, the EFLAGS register contains flags that control the operating mode and state of the processor. Therefore, the PUSHF/PUSHFD instructions prevent the Intel processor from being virtualizable in the same way that the SMSW instruction prevents virtualization. In virtual-8086 mode, the IOPL must equal 3 to use the PUSHF instructions. Of the 32 flags in the EFLAGS register, fourteen are reserved and six are arithmetic flags. The bits of concern are described in Table 2 below.

In contrast to the PUSHF instruction, the POPF instruction allows values in the EFLAGS register to be changed. The effect of the POPF instruction varies based on what mode the processor is operating in. In real-mode, or when operating at CPL 0, all non-reserved flags in the EFLAGS register can be modified except the VM, VIP, and VIF flags. In virtual-8086 mode, the IOPL must equal 3 to use the POPF instructions. The IOPL allows an OS to set the privilege level needed to perform I/O. In virtual-8086

mode, the VM, RF, IOPL, VIP, and VIF flags are unaffected by the POPF instruction. In protected mode, there are several conditions based on privilege levels. First, if the CPL is greater than 0 and less than or equal to the IOPL, all flags can be modified except IOPL, VIP, VIF, and VM. The interrupt flag is altered when the CPL is at least as privileged as the IOPL. Finally, if a POPF/POPFD instruction is executed without enough privilege, an exception is not generated. However, the bits of the EFLAGS register are not changed.

Bit Number	Flag Name	Description
8	TF – Trap	Set to enable single-step mode for debugging.
9	IF – Interrupt Enable	Controls the response of the processor to maskable interrupt requests.
10	DF – Direction	Setting causes string instructions to process addresses from high to low.
12-13	IOPL – I/O Privilege Level	Indicates the I/O privilege level of the currently running task.
14	NT – Nested Task	Set when the current task is linked to the previous task.
16	RF – Resume	Controls the processor's response to debug exceptions.
17	VM – Virtual-8086 Mode	Enables Virtual-8086 mode when set.
18	AC – Alignment Check	Enables alignment checking of memory references.
19	VIF – Virtual Interrupt	Virtual image of the IF flag.
20	VIP – Virtual Interrupt Pending	Indicates whether or not an interrupt is pending.
21	ID – Identification	If a program can set or clear this instruction, the CPUID instruction is supported.

Table 2. Important EFLAGS Register Bits.

The POPF/POPFD instructions also prevent virtualization of the processor. This is because they allow modification of some of the bits in the EFLAGS register that control the operating mode and state of the processor.

B. INSTRUCTIONS THAT REFERENCE THE PROTECTION SYSTEM (RULE 3C)

Many Intel instructions violate rule 3C.

1. LAR, LSL, VERR, VERW Instructions

Four instructions violate the rule in a similar manner: LAR, LSL, VERR, and VERW. The LAR instruction loads access rights from a segment descriptor into a general purpose register. The LSL instruction loads the unscrambled segment limit from the segment descriptor into a general-purpose register. The VERR and VERW instructions verify whether a code or data segment is readable or writable from the current privilege level. The problem with all four of these instructions is that they all perform the following check during their execution: (CPL > DPL) OR (RPL > DPL). This conditional checks to ensure that the current privilege level (located in bits 0 and 1 of the CS register and the SS register) and the requested privilege level (bits 0 and 1 of any segment selector) are both greater than the descriptor privilege level (the privilege level of a segment). This is a problem because a VM normally does not execute at the highest CPL (CPL = 0). It is normally executed at the user or application level (CPL = 3) so that all privileged instructions will cause traps that can be handled by the VMM. However, most operating systems assume that they are operating at the highest privilege level and that they can access any segment descriptor. Therefore, if a VMOS running at CPL = 3 uses any of the four instructions listed above to examine a segment descriptor with a DPL < 3, it is likely that the instruction will not execute properly.

2. POP Instruction

The reason that the POP instruction prevents virtualization is very similar to that mentioned in the previous paragraph. The POP instruction loads a value from the top of the stack to a general-purpose register, memory location, or segment register. However, the POP instruction can not be used to load the CS register since it contains the CPL. A value that is loaded into a segment register must be a valid segment selector. The reason that POP prevents virtualization is because it depends on the value of the CPL. If the SS register is being loaded and the segment selector's RPL and the segment descriptor's

DPL are not equal to the CPL, a general protection exception is raised. Additionally, if the DS, ES, FS, or GS register is being loaded, the segment being pointed to is a nonconforming code segment or data, and the RPL and CPL are greater than the DPL, a general protection exception is raised. As in the previous case, if a VM's CPL is 3 because it is running as a user application in a VMM, these privilege level checks could cause unexpected results if a VMOS assumes that it is in CPL 0.

3. PUSH Instruction

The PUSH instruction also prevents virtualization because it references the protection system. The PUSH instruction allows a general-purpose register, memory location, an immediate value, or a segment register to be pushed onto the stack. This can not be allowed because bits 0 and 1 of the CS and SS register contain the CPL of the current executing task. The following scenario demonstrates why these instructions could cause problems for virtualization. A process that thinks it is running in CPL 0 pushes the CS register to the stack. It then examines the contents of the CS register on the stack to check its CPL. Upon finding that its CPL is not 0, the process may halt.

4. CALL, JMP, INT n, and RET Instructions

The CALL instruction saves procedure linking information to the stack and branches to the procedure given in its destination operand. There are four types of procedure calls: near calls, far calls to the same privilege level, far calls to a different privilege level, and task switches. Near calls and far calls to the same privilege level are not a problem for virtualization. Task switches and far calls to different privilege levels are problems because they involve the CPL, DPL, and RPL of the Intel protection system. If a far call is executed to a different privilege level, the code segment for the procedure being accessed has to be accessed through a call gate. A task uses a different stack for every privilege level. Therefore, when a far call is made to another privilege level, the processor switches to a stack corresponding to the new privilege level of the called procedure. A task switch operates in a manner similar to a call gate. The main difference is that the target operand of the call instruction specifies the segment selector of a task gate instead of a call gate. Both call gates and task gates have many privilege level

checks in their execution that compare the CPL and RPL to DPLs. Since the VM normally operates at user level (CPL 3), these checks will not work correctly when a VMOS tries to access call gates or task gates at CPL 0.

The discussion above on LAR, LSL, VERR, and VERW provides a specific example of how running a CPL 0 operating system as a CPL 3 task could cause a problem. The JMP instruction is similar to the CALL instruction in both the way that it executes and the reasons it prevents virtualization. The main difference between the CALL and the JMP instruction is that the JMP instruction transfers program control to another location in the instruction stream and does not record return information. The INT instruction is also similar to the CALL instruction. The INT n instruction performs a call to the interrupt or exception handler specified by n. INT n does the same thing as a far call made using the CALL instruction except that it pushes the EFLAGS register onto the stack before pushing the return address. The INT instruction references the protection system many times during its execution.

The RET instruction has the opposite effect of the CALL instruction. It transfers program control to a return address that is placed on the stack (normally by a CALL instruction). The RET instruction can be used for three different types of returns: near, far, and inter-privilege-level returns. Much like the CALL instruction, the inter-privilege-level far return examines the privilege levels and access rights of the code and stack segments that are being returned to determine if the operation should be allowed. The DS, ES, FS, and GS segment registers are cleared by the RET instruction if they refer to segments that can not be accessed by the new privilege level. Therefore, RET prevents virtualization because having a CPL of 3 (the VM's privilege level) could cause the DS, ES, FS, and GS registers to not be cleared when they should be. The IRET/IRETD instruction is similar to the RET instruction. The main difference is it returns control from an exception, interrupt handler, or nested task. It prevents virtualization in the same way that the RET instruction does.

5. STR Instruction

Another instruction that references the protection system is the STR instruction. The STR instruction stores the segment selector from the task register into a general-purpose register or memory location. The segment selector that is stored with this instruction points to the task state segment of the currently executing task. This instruction prevents virtualization because it allows a task to examine its requested privilege level (RPL). Every segment selector contains an index into the GDT or LDT, a table indicator, and an RPL. The RPL is represented by bits 0 and 1 of the segment selector. The RPL is an override privilege level that is checked (along with the CPL) to determine if a task can access a segment. The RPL is used to ensure that privileged code cannot access a segment on behalf of an application unless the application also has the privilege to access the segment. This is a problem because a VM does not execute at the highest CPL or RPL (RPL = 0), but at RPL = 3. However, most operating systems assume that they are operating at the highest privilege level and that they can access any segment descriptor. Therefore, if a VM running at a CPL and RPL of 3 uses the STR to store the contents of the task register and then examines the information, it will find that it is not running at the privilege level at which it expects to run.

MOV is another instruction that prevents virtualization of the Intel processor. There are many variants of the move instruction. The only two that prevent virtualization are the two that load and store control registers. The MOV opcode that stores segment registers is a problem because it allows all six of the segment registers to be stored to either a general-purpose register or to a memory location. This is a problem because the CS and SS registers both contain the CPL in bits 0 and 1. Therefore, a task could store the CS or SS in a general-purpose register and examine the contents of that register to find that it is not operating at the correct privilege level. The MOV opcode that loads segment registers does offer a small amount of protection because it does not allow the CS register to be loaded at all. However, if the task tries to load the SS register, several privilege checks occur that once again become a problem when the VM is not operating at the privilege level at which a VMOS is expecting—typically 0.

C. CONCLUSION

The analysis above clearly shows that the Intel processor is not virtualizable according to Goldberg's hardware rules. However, VMware provides a Type II VMM for the Intel Pentium processor. Their product allows a user to run multiple Intel operating systems such as Windows NT and Linux on the same computer at the same time. After installing the VMware product, I am convinced that VMware Inc. does provide a virtualization environment in the form of a Type II VMM. So the question becomes: how does VMware virtualize the Intel processor even though there are seventeen instructions that prevent it from being virtualizable? This question will be addressed in Chapter IV.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. VMWARE ANALYSIS

A. OVERVIEW OF VMWARE

Virtual machine monitors were invented in the 1970s to multiplex expensive mainframe hardware to system programmers. In the past 30 years, hardware performance has increased and prices have decreased, allowing most people to have a PC on their desktops. This trend made VMMs almost non-existent. Today, however, a company called VMware thinks there is a need for VMMs to multiplex and manage complex, expensive software environments. As a result, VMware has created a modern version of the 1970s virtual machine monitor, but for the PC instead of for large mainframes.

Until VMware emerged, it was not possible to run multiple operating systems on the same Intel x86 PC at the same time. Only one operating system at a time could run and a reboot was needed to switch between operating systems. VMware addresses this problem by using what they call "Virtual Platform technology" (patent pending).

The Virtual Platform is a thin software layer that multiplexes the PC's hardware to virtual machines (see Figure 4 below).

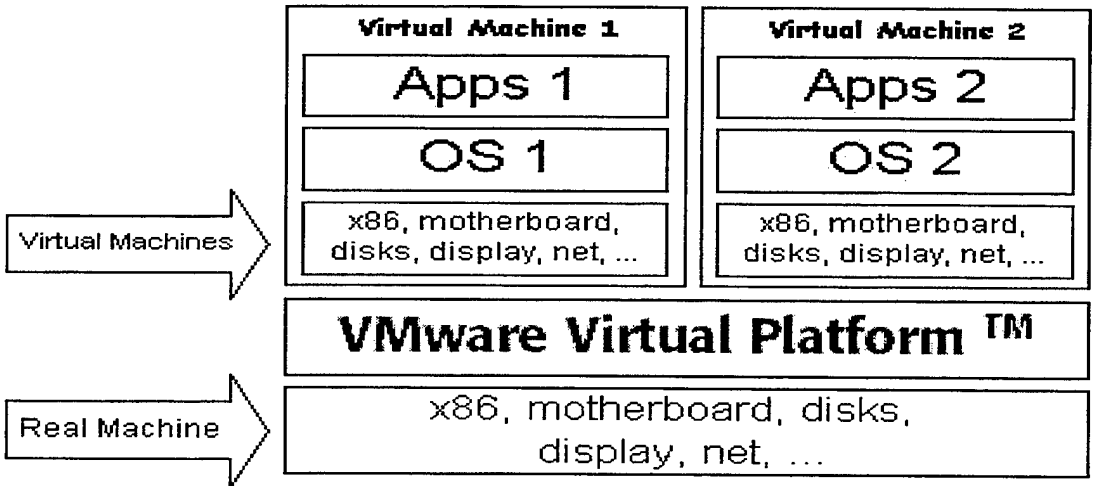


Figure 4. VMware Virtual Platform From Ref. [7].

Each virtual machine can run any operating system the user chooses. VMware for Linux currently supports the following operating systems: MS-DOS 6, Windows 3.1, Windows 95, Windows 98, Windows NT 4.0, Windows 2000 Professional Beta, Red Hat Linux, Caldera OpenLinux, SuSE Linux, and FreeBSD. VMware for Windows NT will support all of the above and also Solaris 7 Intel Edition. VMware only supports these specific operating systems; it can not run any operating system designed for the Intel Pentium architecture. For example, OS/2 and BeOS will not run as VMOSs on VMware. VMware supports the most common types of I/O devices, such as IDE disks, standard floppy drives, Ethernet cards, and sound cards.[3]

1. VMware Claims

VMware Inc. makes many claims about what VMware can do in [Ref. 7]. These claims and an explanation if needed are described below.

Claim: [VMware can] "Run multiple operating systems and their applications simultaneously in separate virtual machines on a standard PC." VMware installs like a normal application in a host OS, either Linux or Windows NT. Running the VMware application allows you to power on a virtual machine. Since the virtual machine has an environment that is very similar to the real machine, many Intel Pentium operating systems and their applications can run inside the virtual machine.

Claim: [VMware can] "Run virtual machine sessions on the X-Windows desktop or in full-screen mode; other virtual machines continue to run in the background." Since virtual machines are really host OS applications, they sit on the host OS desktop. However, if the user toggles the full screen mode in a virtual machine, its VMOS can use the full screen. In full screen mode, it is impossible to tell that other VMs and the host OS are running in the background. A hot key can also be used to switch between virtual machines. Since each virtual machine is a process in the host OS, it is subject to the host

---

[3] MS-DOS, Windows 3.1, Windows 95, Windows 98, Windows NT 4.0, Windows 2000 Professional Beta, and OS/2 are all trademarks of the Microsoft Corporation. All other trademarks including Red Hat Linux, Caldera OpenLinux, SuSE Linux, FreeBSD, Solaris 7 Intel Edition, and BeOS are trademarks of their respective owners.

OS scheduling algorithm. Therefore, all virtual machines that are running will be scheduled like normal applications.

Claim: [VMware can] "Run operating systems already installed on a multi-boot computer without reconfiguring." When configuring VMware to run a virtual machine, a user can specify a mount point for the disk that the VMOS is located on. This allows any OS that is already installed on the hard disk to be used in a virtual machine.

Claim: [Users can] "Install a virtual machine without repartitioning the system's hard drive." If a user would like to create a new virtual machine and there is not enough space left on the hard disk to partition the drive, the user can use a virtual hard disk. If this is done, a virtual machine encapsulates the hard drive for the VMOS and its applications into one file that resides in the host OS file system.

Claim: [Users can] "Encapsulate an entire computing environment and move it between computers as easily as copying a file." Using the example in the previous claim, an entire VMOS and its applications are encapsulated in one file. This file can be copied from one computer to another, allowing an entire computing environment to be moved.

Claim: [VMware permits users to] "Share files and applications among virtual machines using a virtual network within a PC." Each of up to four virtual machines can be assigned a different network address. Each of the virtual machines can also access the network card. This allows virtual machines to share files in many ways. For example, virtual machines could use FTP or NFS to share files.

Claim: [Users can] "Run client-server or Web applications between virtual machines on the same PC." Having multiple network addresses allows a client-server relationship virtual machines. One virtual machine can be designated as the server and another as the client. The client can then use the server's IP address to begin communication.

Claim: [Users can] "Test the same application concurrently on different operating system configurations, i.e. with different amounts of memory, different operating system revisions, or different system settings." Each time a VMware application is started in the

host OS, a VM configuration must be loaded before running a VM. The configuration file can be used to specify different system settings such as those described in the claim. The configuration file can be changed without affecting the hard drive partition or virtual partition on which the VMOS resides. This allows virtual machines with different system configurations to test applications.

Claim: [Users can] "Dedicate a virtual machine to run untrusted applications downloaded from the Internet." Since each virtual machine is isolated from other virtual machines and the host OS, untrusted applications should only affect the virtual machine in which they are running. This claim of isolation will be examined further in Chapter V.

Claim: [Users can] "Upgrade operating system software without losing compatibility-- the legacy OS and its applications are simply transferred to a virtual machine." This claim is self-explanatory.

Claim: [Users can] "Rely on a known stable hardware platform, defined by VMware Virtual Platform. Virtual machines configured for this stable hardware platform will correctly execute on any hardware that supports the virtual platform." When a VMware user defines a configuration for a virtual machine, he specifies many things such as how much memory the VM will have, whether it will have a network card, etc. VMware abstracts the real hardware and presents a specific hardware platform to each VMOS. For example, 10 different computers could have 10 different network cards. However, all VMOSs are configured for the same type of network card, as defined by VMware. Therefore, virtual machines that are configured to run with a network card will run on any VMware virtual machine, no matter what the real network card in the machine is.

2. Virtual Platform Architecture

VMware works in conjunction with a host operating system. The OS can be either Linux or Windows NT. VMware has a "dual-mode personality" which is illustrated in Figure 5. VMware runs both as an application in the host operating system and as a VMM running directly on the hardware. VMware implements as much as possible in the VMM because, since it sits directly on the hardware, it provides greater

performance than the application portion of VMware. The application portion of VMware is used for the device-dependent portions of VMware. VMware did not have to write their own device drivers because they simply use the device drivers of the host operating system.

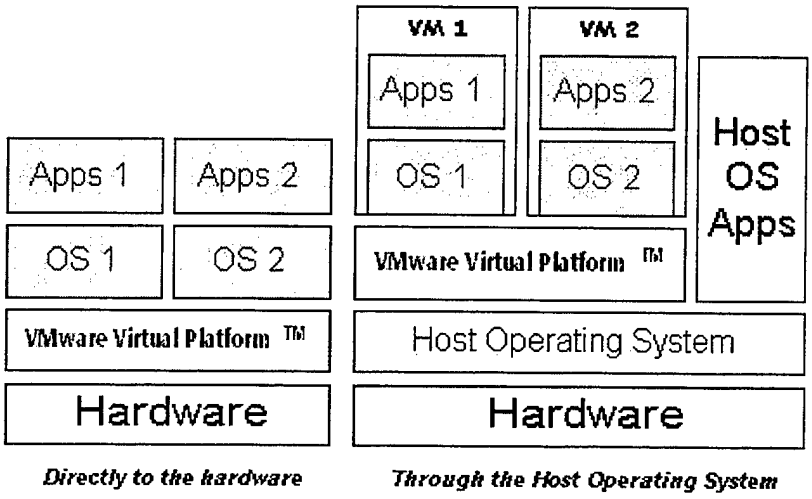


Figure 5. VMware Virtual Platform Dual Mode Personality From Ref. [7].

VMware Virtual Platform uses three software components to implement the "dual-personality" described above. They are the application, monitor, and driver. These components and their relationships are illustrated in Figure 6 below.

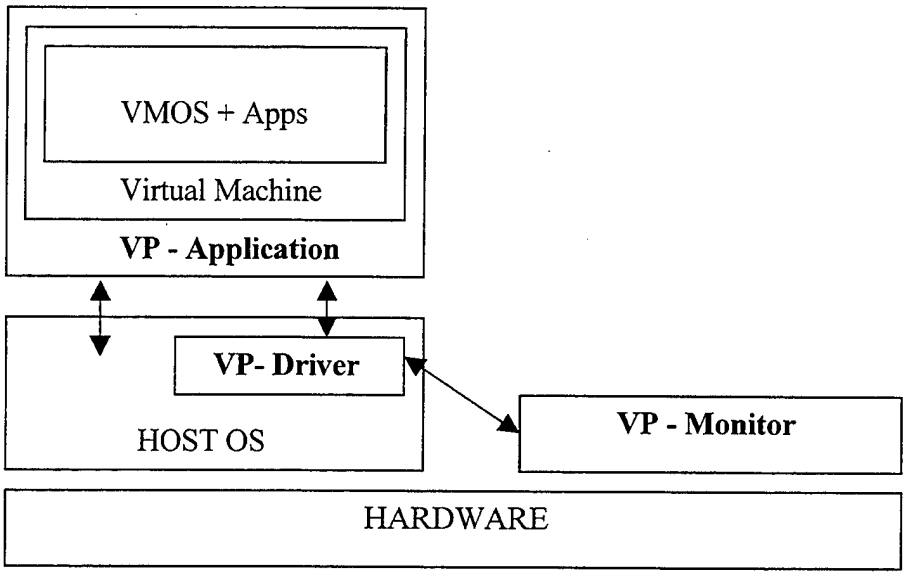


Figure 6. VMware Virtual Platform Architecture.

The driver component is a device driver written for the host operating system. It provides a tailored interprocess communication mechanism between the monitor and the application. The host operating system is unaware that VMware's monitor interacts directly with the PC's hardware. Additionally, the Virtual Platform runs as a normal application process under the host operating system. The driver is only necessary because the host OS is protected from user applications. If the host OS were not protected from user applications, the application and monitor could communicate directly.

The monitor component runs in privileged mode directly on the hardware, allowing it to execute privileged instructions without appealing to the host operating system. Without the monitor, VMware would resemble a simulator with much lower performance. The monitor calls the application component through the device driver to access system resources, including processor scheduling, physical memory management, and device drivers.

The application component is installed like a normal application on the host OS. It configures, launches, and administers virtual machines. After being called by the monitor (through the device driver) to access system resources, the application makes the appropriate calls to the host OS.

3. Isolation

VMware uses Intel hardware protection mechanisms to isolate virtual machines from the host operating system and from each other. VMware says that the isolation "does not make any assumptions concerning the software that runs within the virtual machine. Even a rogue application or operating system is confined to the VMware Virtual Platform" [Ref. 7]. In fact, the VMware product specification states that virtual machines are isolated from faults.

Since virtual machines are not supposed to affect each other or the host OS, crashes that occur in a virtual machine should not affect data or applications outside of the crashing virtual machine. However, VMware does admit that any operating system that uses an undocumented or undefined feature of the hardware could be incompatible.

Therefore, operating systems that rely on undocumented or unknown features of the hardware may not work correctly, but such operating systems can not crash VMware or interfere with other virtual machines.

4. Performance

Without high performance, VMware would not be a worthwhile product. By directly executing instructions on the processor, VMware reduces performance overhead. VMware claims that the "overhead of VMware Virtual Platform can be as low as 3%-5% for certain computation-intensive benchmarks" [Ref. 7]. Most applications running in virtual machines perform as if they were running on the real machine. Performance is the main difference between VMware and traditional simulators and emulators.

B. VMWARE AS A "VIRTUAL VMM"

After analyzing VMware literature and installing and running the VMware product, it is evident that VMware has developed a product that is close to a Type II VMM for the Intel Pentium architecture. Screen shots from VMware running with both a Linux host operating system and a Windows NT host operating system are on the following page.

Even though the VMware product works very much like a Type II VMM, there are many reasons why VMware should be considered a "virtual VMM" and not a true Type II VMM. These reasons can be divided into two categories: hardware and software.

1. Hardware

The first hardware reason that VMware is not a true Type II VMM is that, as mentioned in the previous chapter, the Intel processor is not truly virtualizable because of the seventeen sensitive, unprivileged instructions. Therefore, in order to implement a VMM on the Intel architecture each of these instructions has to be emulated by software in a VMM. However, emulating the instructions is not the hardest part. Since these instructions are not privileged, they will not trap when executed by user-level code. This means that the VMM must examine instructions before they are executed to see whether an instruction is one of the seventeen problem instructions that will not generate a trap.

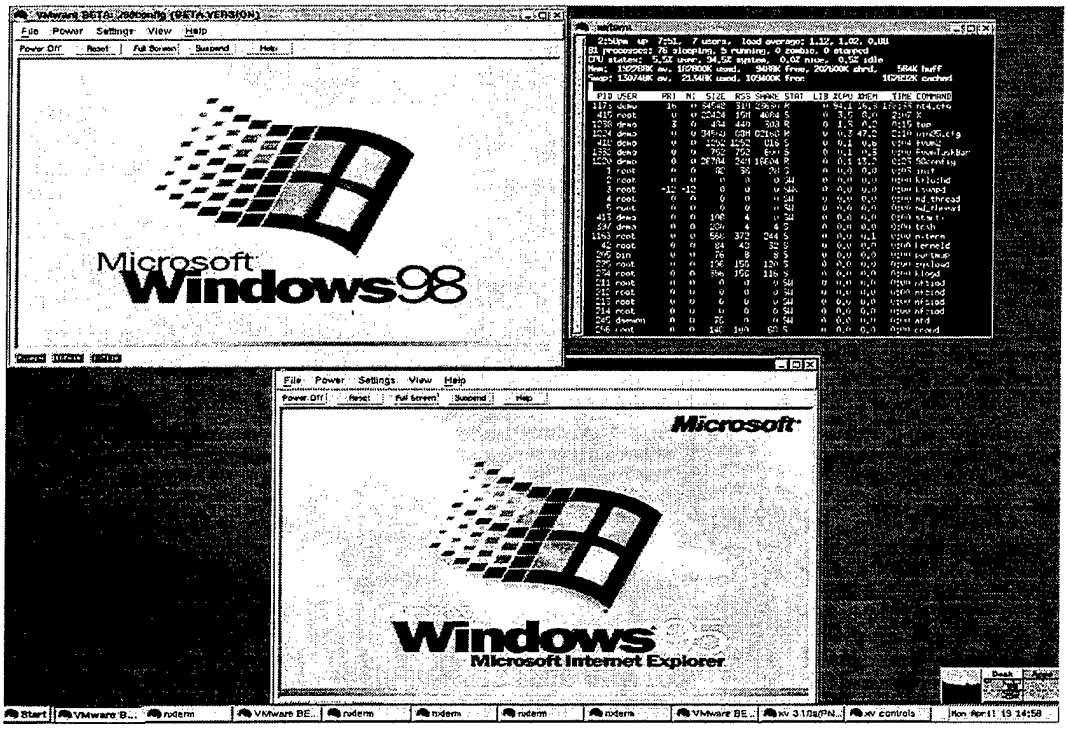


Figure 7. VMware Screen Shot Using Linux Host OS From Ref. [7].

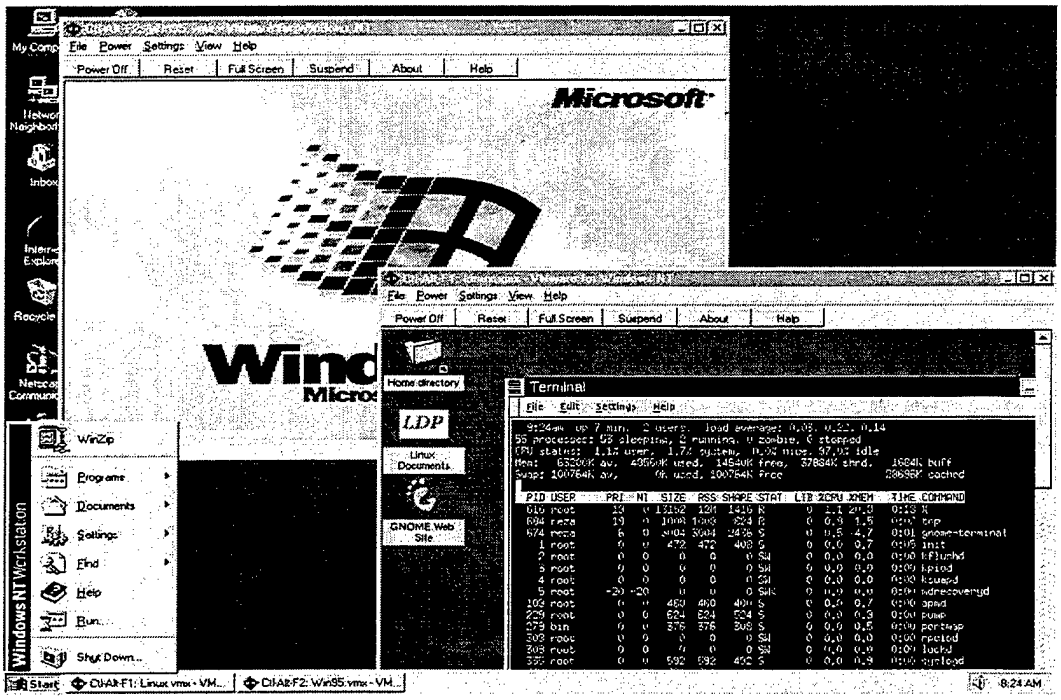


Figure 8. VMware Screen Shot Using Linux Host OS From Ref. [7].

The seventeen "problem" instructions are not the only roadblock for virtualizing the Intel processor. Since the processor uses segmentation and rings in its design, it is harder to virtualize because any VMM desiring to virtualize the processor must also virtualize these features. One such problem is known as ring collapsing [Ref. 1]. A VMM prevents access to the real system hardware by not running VM ring 0 code as ring 0 code in the real system. This means that ring 0 of the VM must be mapped to a less privileged ring. The rings in the Intel architecture are both ordered and finite. There are many schemes that can be used to solve the ring collapsing problem. The easiest is to map ring 0 of the VM into the VMM, meaning that any ring 0 instruction is emulated by the VMM. This is, in effect, a hybrid VMM. To avoid the poor performance of an HVM, another strategy is to map two virtual, adjacent rings into the same physical ring. This technique has the side effect of destroying the ring boundary between the two adjacent layers that are used. Fortunately, most operating systems written for the Intel processor only use two rings of the processor: ring 0 and ring 3 for the operating system and applications respectively. This simplifies the ring collapsing problem because a VMM would have three physical rings in which to emulate two virtual rings.

2. Software

The first software reason why VMware is not a true Type II VMM is that VMware can not run some operating systems designed for Intel Pentium processors. If VMware provided true virtual environments that are duplicates of the real machine, it should be able to run any operating system that can run on the real machine. However, when this thesis was written, VMware could not support the Windows 2000 Professional Beta or the Solaris 7 Intel Edition as a guest operating system. VMware does plan to support these operating systems in the future. In addition, there are many other operating systems that are known not to work as VMware guest operating systems and will not be supported by VMware. These include BeOS, Minix, OS/2, OS/2 Warp, QNX, SCO Unix, and SCO Unixware. It is not clear why these operating systems will not run in a VMware virtual machine. VMware does say that any operating system that relies on undocumented or undefined features of the PC hardware will likely be incompatible with

their product. However, the reason that these guest operating systems will not run in a virtual machine probably has more to do with special idiosyncrasies in their implementation than with undocumented or undefined instructions and features. Furthermore, VMware may not support these operating systems as guest OSs because their market share is so low that it is not cost effective to test and guarantee their operation.

A second reason why VMware is not a Type II VMM is because the Intel Pentium architecture does not meet Popek's essential VMM requirement. As stated in [Ref. 3], "a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions." For the Intel architecture, I have demonstrated that this is not the case in the previous chapter.

Finally, Popek's characteristics of a VMM are: 1) the VMM provides an environment that is "essentially identical" to the environment of the real machine, 2) programs running in this environment show only minor decreases in speed, and 3) the VMM is in complete control of system resources. Popek described "essentially identical" as being a program running exactly the same as if it were run on the real machine, except for possible timing dependencies or system resource availability. VMware does not have this characteristic because a VMOS that runs under VMware may not work correctly if it is configured to support the real hardware attached to the underlying real machine. In order to make Windows detect hardware correctly in the VM environment, the system administrator must install the driver that VMware tells him or her to. For example, even though the computer has a 3COM NIC card, the driver specified to Windows may be different. Another example of VMware not providing an identical environment is that parallel ports are only uni-directional in the VM environment. This means that parallel external Iomega Zip and Imation Superdisk drives can not be used.

V. CAN AN INTEL "VMM" BE SECURE?

In this chapter, I will examine several security issues for a VMM that is designed for the Intel Pentium architecture. The chapter begins with a discussion on the VAX Security Kernel to show that secure VMMs are possible. Second, security of the Intel processor is covered. Third, security issues that arise when using the Intel architecture to run VMMs are discussed. Fourth, security ramifications of using VMware to separate mandatory security classes are examined. Finally, a better approach for using a VMM to separate mandatory security classes on the Intel architecture is covered.

A. ARE SECURE VMMS POSSIBLE?

Before discussing whether or not the Intel processor can support a secure VMM, it is necessary to show that secure VMMs are possible. A highly secure Type I VMM was the VAX Security Kernel [Ref. 8]. The VAX Security Kernel was created to develop a Type I virtual machine monitor for the VAX architecture. The system's hardware, microcode, and software were designed to meet TCSEC Class A1 assurance and security requirements. The project also maintained standard VMS and Ultrix-32 interfaces to run COTS operating systems and applications in virtual machines.

A security kernel is defined as hardware and software that implements the reference monitor concept. [Ref. 9] A reference monitor enforces authorized access relationships between the subjects and objects of a system. An implementation of a reference monitor is called a reference validation mechanism. Three design requirements must be met by a reference validation mechanism:

1) The mechanism must be tamperproof.
2) The mechanism must always be invoked.
3) The mechanism must be small enough to be to subject to analysis and tests to ensure completeness.

The VAX security kernel is a VMM that allows multiple virtual machines to run concurrently on a single VAX system. The security kernel could support a large number of simultaneous users and provided isolation and controlled sharing of sensitive data. Since the VMM was a security kernel, many traditional features of VMMs, such as self-virtualization and debugging of one VM from another were not implemented in order to reduce kernel complexity and make the kernel highly secure.

The VAX processor, much like the Intel Pentium processor, contained several instructions that were sensitive but not privileged. It also has four rings like the Intel processor. Since this project was conducted by Digital Equipment Corporation, the manufacturers of the VAX processor, the security kernel designers had the luxury of modifying the VAX processor microcode in order to make it virtualizable. Hall et al. [Ref. 10] describe the four instructions that prevented virtualization on the VAX processor: CHM, REI, MOVPSL, and PROBE. The CHM instruction switches to a mode of equal or increased privilege. The REI instruction switches to a mode of equal or decreased privilege. The MOVPSL instruction is used to read the Processor Status Longword (similar to the machine status word in the Intel architecture). The PROBE instruction is used to determine the accessibility of a page of memory. These four instructions read or write one of the following pieces of sensitive data: the current execution mode, the previous execution mode, the modify bit of a page table entry, and the protection bit of a page table entry. With only four problem instructions, the VAX processor did not have as many virtualization problems as the Intel processor does.

In order to make the VAX processor virtualizable, DEC did not want to simply modify sensitive, unprivileged instructions to be privileged because standard VAX operating systems would not be able to run. An important project goal was to be able to run standard operating systems on the processor that supported the VMM in order to provide VMM users and the DEC sales force with flexibility in choosing their hardware and software. Furthermore, customers were already familiar with and had applications for VMS and Ultrix-32. Therefore, DEC used a more sophisticated approach. Some of

the changes they made included defining a new VM mode bit, defining a new register called VMPSL, and defining a VM-emulation exception. DEC also changed microcode in order to address the four problem instructions described above.

DEC used ring compression to avoid some modifications to the processor. Ring compression was implemented entirely in software and maps rings 0 and 1 (kernel and executive) of the VM to ring 1. This breaks down the protection between these two layers but was thought to be the simplest and most secure way to virtualize the four rings of the VAX processor. This choice had little security impact since, although the VMS operating system for the VAX used all four rings, all three inner rings were in fact used for fully trusted operating system software.

Some virtual machine monitors (including, especially, IBM's VM/370) virtualize not only the CPU, but also the I/O hardware. Virtualizing the I/O hardware allows a VMOS to run almost unmodified. The VAX I/O hardware was difficult to virtualize because its I/O mechanisms read and write various control and status registers in the I/O space of physical memory. To overcome this difficulty, the VAX security kernel I/O interface used a special kernel call mechanism that was optimized for performance. To use this mechanism, a virtual machine executed a Move To Privileged Register (MTPR) instruction to a special kernel call register. The MTPR instruction trapped to software in the security kernel that performed the I/O. This new I/O interface meant that untrusted device drivers had to be written for all VMOSs that would run under the VMM. In choosing this strategy, the VAX security kernel development team was guided by the goal of making the operating system developer's task of supporting a virtual machine comparable to the task of supporting any other new VAX processor. Since each new VAX processor required some I/O interface and driver support by each operating system, the addition of a kernel call I/O interface was felt to be a reasonable design choice.

The VAX security kernel applies mandatory and discretionary access controls to virtual machines. The kernel assigns every virtual machine an access class consisting of a secrecy class (based on the Bell and LaPadula model) and an integrity class (based on

the Biba model). The kernel also supports access control lists on all objects including real devices, disk and tape volumes, and security kernel volumes. The VMM security kernel is very different from a typical secure operating system because the subjects and objects are virtual machines and virtual disks, not files and processes. Files and processes are implemented by each VMOS.

The two kinds of subjects in the VAX security kernel are users and virtual machines. Users communicate through a trusted path to a server process. The server processes are trusted and run only within the security kernel. The users minimum and maximum access class, the terminal's minimum and maximum access class, the user's discretionary access rights and privileges, and the privileges exercisable from the terminal all determine what the server can provide to the user. The other type of subjects, VMs, are untrusted subjects that run a VMOS. The VMOS is operated normally and will not affect the security of other virtual machines or the security kernel even if the operating system is penetrated.

When a user logs into the security kernel, the VMM establishes a session between the user's terminal line and the user's server. To connect to one of the VMs, the user issues a CONNECT command that specifies the name of the VM. If the connection is authorized, the security kernel suspends the session with the server and establishes a session between the user and the requested virtual machine. Virtual machines can be set up to run single users or multiple users.

The VAX security kernel supports three types of objects: real devices, disk and tape volumes, and security kernel volumes. Real devices include those that can contain or transmit information and must be controlled by the TCB. These include disk drives, tapes, printers, terminal lines, and network lines. The contents of some disk and tape volumes are controlled completely by a virtual machine. Other disk volumes have a VAX security kernel file structure and can not be directly accessed by a VMOS. These are called VAX security kernel volumes. VAX security kernel volumes contain VAX security kernel files that are organized as a flat file system. These files are used for many

things including the implementation of virtual disk volumes for use by virtual machines and storage of long-term system databases such as the audit log and the authorization file.

Since the VAX security kernel was designed to meet Class A1 requirements, it had to enforce mandatory access controls. Each kernel object is assigned a sensitivity label, called an access class, consisting of a secrecy and an integrity class. Both of these classes are subdivided into a hierarchical level and a category set. The kernel supports 256 secrecy levels, 256 integrity levels, 64 secrecy categories, and 64 integrity categories.

To design the VAX security kernel, levels of abstraction were used to help reduce complexity and make specifications more precise and understandable. A layer only depends on layers below it. This prevents deadlock by removing loops [Ref. 11]. The layered design also allowed the kernel developers and testers to analyze the correctness and security of each layer independently and to be sure that there were no complex interactions between kernel components that might jeopardize system security. Table 3 contains the name of each layer and a brief description from [Ref. 8].

The designers of the VMM security kernel realized that highly-secure systems are often hard to use because of their limited interfaces. Interface features are large and hard to verify, preventing them from being included in the kernel. To overcome this problem, the designers created two separate command sets: server commands and administrative commands. The secure server commands are all implemented in trusted code. These commands include those that control terminal connections to virtual machines. Some of these commands are CONNECT, DISCONNECT, RESUME, and SHOW SESSIONS. The SECURE, or administrative commands, are commands that help manage the system. These commands are issued and parsed in a VMOS.

A major goal of this development effort was to receive a Class A1 rating under the TCSEC. In addition, the designers were also interested in achieving good overall performance and compatibility with a large amount of existing software. This second goal was motivated because no Class A1 system, including Honeywell's STOP kernel for the SCOMP [Ref. 4] and Gemini Computers' GEMSOS [Ref. 5], had ever been built that

was compatible with an existing body of application software. Even after the modification of the processor microcode, the VAX security kernel only allowed virtual machines to run at 47-48% of their performance on a real machine. Although the project was canceled before many performance enhancements could be tested, DEC admitted that major improvements in performance would be difficult without modifying the microcode of the processor or the operating systems that run in the virtual machine [Ref. 10].

DEC's efforts did lead to several conclusions about virtualization:

1) Every ring of a processor can be emulated, but this is often not necessary.
2) Emulating a start I/O instruction is simpler and cheaper than emulating memory-mapped I/O.
3) Defining the VM as a particular processor or family of processors makes the VM more portable than if it were a reflection of the actual hardware. For example, if a VM is defined to be a Pentium processor, the VM will work on a Pentium II or Pentium III processor.
4) Performance of a VM suffers when sensitive instructions must be made to trap to emulation software.
5) There are alternatives to modifying the microcode support for every privileged instruction to meet the needs of the VMM.
6) If a VMM is a security kernel, handshakes between the VMM and VM's must be scrutinized because the VMM can not trust the VM operating systems.

These conclusions should be considered in any attempt to design a secure Type I VMM for the Intel Pentium architecture.

B. HOW SECURE IS THE INTEL ARCHITECTURE?

The 80x86 architecture (including Intel and clones) was clearly the choice for trusted systems during the 1980's and early 1990's. Some of the high-assurance trusted products that used the 80x86 processors were the Boeing MLS Lan (A1) [Ref. 13], Gemini Trusted Network Processor (A1) [Ref. 5], Verdix VSLAN (B2) [Ref. 14], TIS Trusted

Layer Name	Definition
Users	Includes untrusted application programs that run on the VMOS and humans who communicate with the secure server through the trusted path
VMOS	The virtual machine's operating system
Secure Server	Implements the trusted path for the security kernel, logs users in and out, provides security-related administrative functions; everything above this layer is untrusted
Virtual VAX	Completes the virtualization process by emulating sensitive instructions and delivering interrupts and exceptions to the virtual machine
Kernel Interface	Implements virtual controllers for virtual I/O devices and the security function controller
Virtual Printers	Implements virtual printers for each VM and multiplexes the real physical printers among virtual printers
Virtual Terminals	Implements virtual terminals for each VM and manages physical terminal lines
Volumes	Implements VAX security kernel and exchangeable volumes and provides registries of all subjects and objects
Files-11 Files	Implements a subset of the ODS-2 file system [4]
Auditing	Provides the facilities for security auditing and alarms
Higher-Level Scheduler	Creates the abstraction of level-two virtual processors (vp2s); dedicated vp2s are used by the Secure Server layer and bindable vp2s are used by virtual machines
VM Virtual Memory	Implements the shadow page tables that are needed to support virtual memory in virtual machines
VM Physical Memory	Manages real physical memory and assigns it to virtual machines
I/O Services	Implements device drivers that control the real I/O devices
Lower-Level Scheduler	Creates the abstraction of level-one virtual processors (vp1s) that are the basic unit of scheduling for the system; vp1s are intended to be very inexpensive processes for use within the kernel
Hardware-Interrupt Handler	Immediately above the VAX hardware and modified microcode; contains the interrupt handlers for the various I/O controllers and certain CPU-specific code

Table 3. Layers In The VAX Security Kernel Design.

[4] A summary of the Files-11 ODS-2 structure can be found in the appendices of a DEC publication [Ref. 12].

Xenix (B2) [Ref. 15], and the XTS-300 (B3) [Ref. 4].

1. Hardware Rings

The four rings of the Intel architecture are a benefit to secure development. Rings are a generalization of the original supervisor/user design for hardware. Some of the benefits of hardware rings as described in [Ref. 16] are: users can create arbitrary, protected subsystems that can be used by others; system or kernel code can be implemented in layers that are enforced by the hardware; and the user can protect himself while debugging his own programs. Creating protected subsystems allows a reference monitor as described by Anderson in [Ref. 17]. Any secure VMM must be both tamperproof and non-bypassable. Since the Intel processor protects operating system code, one can be assured that a VMM can not be tampered with. Additionally, if a VMM is designed to be non-bypassable, one is assured that the VMM will always be invoked.

The second benefit of rings is that a layered kernel can be implemented with hardware rings. A layered kernel has many security advantages. It limits the propagation of errors, making a secure implementation easier to achieve. For example, changes in ring 1 of a TCB or OS would not effect the kernel that has already been certified at ring 0. However, for performance reasons, all layers of a kernel are normally in ring 0.

2. Operating System Design Issues

Typically, much more emphasis has been placed on assurance of a system's software components rather than its hardware components. The hardware components of a system are often assumed to operate securely if they are used correctly. However, this may not always be the case. This section discusses "pitfalls" of the Intel architecture that were discovered by Sibert et al. in [Ref. 18]. System designers for the Intel architecture should consider these when implementing secure operating systems because they can lead to security problems if handled incorrectly.

As a result, Sibert et al [Ref. 18] conducted an in-depth analysis of the 80x86 processor families (included Intel and other clones) to look for any architectural properties that could have "unexpected, and undesirable, results in a secure computer system." Although the authors did not find any "gross security flaws," they did find

several features that introduce previously unreported covert channels and other problems in the Intel architecture.

One potential security problem that the authors found is undocumented instructions. At the time of their examination, Intel had an Appendix H to their programming manual that contained functions that were explicitly undocumented and available only under strict non-disclosure protection. It was not possible for the authors to test these undocumented features.

The architectural "pitfalls" that were identified in this paper are not security flaws in the processor. The pitfalls are problems that can cause security flaws in a system if the operating system does not correctly handle the pitfalls. For example, in many cases, if a TCB virtualized the register that causes a pitfall, a covert channel would be closed. Most of the pitfalls that were discussed allow possible covert channels, both timing and storage. A covert channel is a mechanism used to transfer information from one process to another that is not intended to transfer information [Ref. 19]. If a covert channel exists, a high subject can signal a low subject and bypass the security policy enforcement mechanism. These pitfalls included the TS flag, FPU context, segment accessed bit, segment attributes, page access visibility, internal register visibility, debug register values, time stamp counter, performance counter, cache and TLB, and undefined values. Sibert et al. discussed each of these pitfalls in [Ref. 18]. Many of these pitfalls were identified in very early versions of the Intel processor such as the 386 and 486. However, most of these pitfalls still exist in the current Pentium architecture because Intel has remained backward compatible all the way to the 8086, a processor developed in the late 1970's.

An example of a pitfall that still remains today is the TS bit of control register 0. The SMSW instruction stores the lower 16 bits of control register 0 and makes the TF flag visible. Sibert et al describe a detailed example of a channel using the TF flag.

A variation of this pitfall is also still present in the Intel Pentium architecture. As mentioned, the TS bit is visible to unprivileged tasks. However, even if the bit was not

visible, the state of the bit could be determined by the duration of a floating point instruction. The TS flag helps minimize the number of times the FPU state must be saved. If the current task was the last task to use the FPU, the state of the FPU does not need to be saved. However, if a new task is using the FPU, the state of the FPU must be saved and the FPU is re-loaded from the current task's FPU context. If the FPU is saved and then re-loaded, it will take much longer than if these actions were not necessary. Thus, information flow is possible between two tasks because of the optimization of the FPU context-saving process.

In addition to the pitfalls just described, Sibert et al. [Ref. 18] also describe some of the 102 reported flaws on different versions of the Intel 80x86 architecture. Most of these flaws would probably not translate into exploitable security flaws. However, some of the flaws did. The authors gathered 102 reports of flaws, 17 of which are security relevant. Nine of the flaws were denial of service flaws that could halt the processor. All of the flaws were found on the Intel Pentium and earlier processors. The analysis showed that Intel did fix most of these problems in each subsequent processor release. However, current Intel processors may have new flaws that are security relevant.

C. THE SECURITY OF INTEL VIRTUALIZATION

Since the Intel Pentium architecture is not truly virtualizable, a bit of "trickery" is required in order to make a "virtual" VMM run on the architecture. The trickery is required to detect any instructions that are sensitive but not privileged before they are executed by a VM. Several different techniques could be used to accomplish this. I will describe the techniques that Lawton has proposed [Ref. 20].

1. Pure Emulation

The first technique is pure emulation. Emulation is a technique that maps one system architecture into another system architecture. By modeling a large part of the x86 instruction set in software, emulation allows x86 operating systems and applications to run on non-x86 platforms. Lawton is the leader of a software project called Bochs [Ref. 21]. Bochs emulates a majority of the x86 CPU, related AT hardware, and a BIOS and is able to

run DOS, Windows '95, Minix 2.0, and other x86 operating systems on both x86 and non-x86 platforms. Some of the architectures that Bochs can run on include Sparc, PowerMac, SGI, and x86. Bochs is able to run approximately 1.5MIPS on a 400Mhz PII Linux machine. Bochs is closer to a complete software interpreter machine than a VMM. The disadvantage of using this technique to virtualize the Intel architecture is significant performance degradation since no instructions are ever executed directly on the hardware. There are some advanced techniques, such as dynamic translation, which can improve performance. Dynamic translation allows sequences of small, Intel architecture code to be translated into native-CPU code "on-the-fly." Since the native code is cached, it can run significantly faster. However, the performance will never achieve that of a Type I VMM.

2. OS/API Emulation

A second technique proposed by Lawton is OS/API emulation [Ref. 20]. Applications normally communicate with an operating system with a set of APIs. OS/API emulation involves intercepting and emulating the behavior of the APIs using mechanisms in the underlying operating system. This allows applications designed for other x86 operating systems to be run. This strategy is used in a project called Wine. Wine is "an implementation of the Windows 3.x and Win32 API on top of X and Unix" [Ref. 22]. Wine has a program loader that allows unmodified Windows 3.1/95/NT binary files to run on Intel x86-based Unix machines, such as Linux, FreeBSD, and Solaris. Wine allows application binaries files to run natively, meaning that Windows executable files can be run in Unix environment without modification. Running binaries natively allows better performance than the pure emulation technique described above. However, OS/API emulation only works on members of the x86 OS family for which the APIs have been emulated. Furthermore, OS/API emulation is very complex. A VMM is less complicated and requires less change as an OS evolves from release to release.

3. Virtualization

A third technique is virtualization. Most hardware is only designed to be driven by one device driver. The Intel Pentium CPU is not an exception to this rule. It is designed to be configured and used by only one operating system. Features and

instructions of the processor designed for applications are generally not a problem for virtualization and can be executed directly by the processor. A majority of a processor's load comes from these types of instructions. However, certain sensitive instructions are not privileged in the Intel architecture, making it very difficult for a VMM to detect when the instructions are executed. A strategy for virtualizing the Intel architecture would be as follows[5]:

1) Non-sensitive, unprivileged application instructions can be executed directly on the processor with no VMM intervention.
2) Sensitive, privileged instructions will be detected when they trap after being executed in user mode. The trap should be delivered to the VMM that will emulate the expected behavior of the instruction in software.
3) Sensitive, unprivileged instructions present a problem because the processor does not offer natural hardware protection against them. They must be detected so that control can be transferred to the VMM.

The hardest part of the virtualization strategy described above is protecting against the seventeen problem instructions that were described in Chapter III. Lawton describes how this is accomplished for FreeMWare [Ref. 20]. It analyzes instructions until one of the following conditions is encountered:

1) A problem instruction.
2) A branch instruction.
3) The address of an instruction sequence that has already been parsed.

If case 1 or 2 is encountered, a breakpoint must be set at the beginning of the problem or branch instruction. If case 3 is encountered, execution continues as normal since this code has been analyzed already and necessary breakpoints have been installed.

---

5 This is probably the strategy that VMware used in order to virtualize the Intel architecture. Lawton, the author of the Bochs software, is also leading an effort to create an open source version of VMware. The title of this project is called FreeMWare [Ref. 20].

Code is allowed to run natively on the processor and it continues to run until it reaches a breakpoint. If the breakpoint occurred because of a problem instruction, its behavior is emulated by the VMM. If the breakpoint occurred because of a branch instruction, it is necessary to single step through its execution and begin analyzing instructions again at the branch target address. If the target address is not computed and has already been analyzed and marked as safe, then the branch instruction can also be marked as safe and it can run natively on the processor on subsequent accesses. Computed branch addresses require special attention. These instructions must be dynamically monitored to ensure that execution does not branch to code that has not been analyzed. A table might be used to keep track of the breakpoints. It might include the type of condition for which the breakpoint was set: a problem instruction, a branch instruction, or the address of an instruction that has already been parsed.

Lawton's strategy also accounts for the possibility that some instructions may write into memory, possibly into the address of instructions that have already been analyzed and marked as safe. The paging system is used to prevent this by write protecting any page of memory in the page tables that has already been analyzed and marked as safe. All page entries that point to the physical page with analyzed code would have to be write protected since multiple linear addresses can be mapped to the same physical page. When a write-protect page fault occurs, this gives the VMM the opportunity to unprotect the page and step through the instructions. If a problematic instruction is written into a page while stepping through instructions, a breakpoint should be installed before that instruction. Finally, the page should be write-protected again. In cases where instructions cross a page boundary, both pages are write-protected and the modified code will be handled in both pages. Tables are used to track which instructions have been analyzed. The tables use the page size for performance reasons. Lawton describes more details of how FreeMWare is being implemented in his paper [Ref. 20]. Some of the issues discussed include: pass-through I/O devices, timing issues, virtualizing descriptor loading. Each of these are described below:

a. Pass-Through I/O Devices

A guest OS can only use devices that are provided to it by the virtualization environment. Thus, a guest OS will not be able to use hardware that is not supported by a host OS driver. However, it may be useful to allow a device driver in the guest OS to drive hardware for a device that is not supported by the host OS. For example, a Linux host OS will not support a Winmodem. This means that a VM running Windows that runs Linux as its host OS can not use a Winmodem since it is not supported by the host OS. A solution to this is called pass-through devices. Pass-through devices could allow a guest OS to communicate with devices using a pass-through mechanism that handles I/O reads and writes. It is difficult to implement pass-through I/O devices securely because control of the real hardware control is turned over to the VMOS.

b. Timing

A VMM for the Intel architecture must also consider timing issues. The VMM must accurately emulate the system timers. Every time slice of native code execution is bounded by an exception that is generated by the system timer. This exception means that the execution time slice is over. The exception vectors to a routine that is defined in the VMM's IDT for a guest OS. A mechanism is needed that measures the time between these exceptions to emulate an accurate timer. On Intel Pentium processors and later, performance monitoring could be used. The RDTSC, Read Time Stamp Counter, instruction gives an accurate time stamp reading that can be used. The instruction is also executable in CPL 3, allowing efficient use in user-level VMM code.

c. Virtualizing Descriptor Loading

A VMM for the Intel architecture must also have its own set of LDT, GDT, and IDT tables. These are necessary for two reasons. First, it allows the segment register mechanisms to work naturally. Second, it allows the VMM to have its own set of exception handlers. A time slice is up when the system timer invokes a routine in the private IDT. When this occurs, the VMM must revert back to the tables that are used by the host OS, meaning that the host OS is not aware of the virtualization environment.

Since all privilege levels (0-3) in a VM are mapped into CPL 3, the CPL is not high enough when trying to load code that is less than CPL 3. CPL 3 code can load descriptors as expected as long as the GDTR and LDTR registers point to the guest OS's descriptor tables. When running system code in CPL 3, exceptions are generated when loading a descriptor with that has CPL < 3. This does not occur when system code is executed at CPL 0 as it expects. In order to solve this problem, one must trap and emulate instructions that load the segment registers when running at CPL < 3. One must also virtualize all instructions that examine segment registers with PL < 3 because they may look at the RPL field which will not reflect the expected privilege level.

Another technique that will help solve this problem is the use of a private GDT and LDT for the virtualization of code at CPL < 3. Since, the instructions that reference the GDTR and LDTR are emulated, they can be loaded with values that point to the private GDT and LDT. The private descriptor tables would start out empty and generate exceptions when a segment register loads. Each time this happens, a private descriptor is generated that allows the next segment register load to execute natively. Every time the GDTR and LDTR are reloaded, the private descriptor tables are cleared.

D. ADDITIONAL INTEL VIRTUALIZATION INSIGHTS

In addition to the details of the FreeMWare project, another useful resource on how Intel virtualization may be accomplished is the Disco prototype, mentioned in the introduction to this thesis. Disco was developed at Stanford University on the Stanford FLASH shared-memory multi-processor. [Ref. 2] The Disco project is an implementation of a Type I VMM for the Flash multi-processor. It runs several different commercial operating systems in virtual machines to provide high-performance system software. Much like VMware, Disco was able to overcome the high overhead and poor resource sharing that is typical of both Type I and Type II VMMs. Some of the key insights of the Disco implementation that would be applicable to virtualizing the Intel Pentium architecture are described here.

1. Virtual CPUs

Since most machines only use one processor, multiple VM's running on the same hardware each have a virtual processor. This is an abstraction to make each VM and its associated VMOS think that it has the sole use of the real processor. In order to schedule a virtual CPU for a VM, the VMM needs to set the real machine's registers to those of the virtual CPU and jump to the program counter of the virtual CPU. A data structure is kept for each virtual CPU that contains register contents, TLB contents, and other state information of the virtual CPU when it is not running on the real CPU. If a virtual CPU is running on the real CPU, a VM trap, such as a page fault, system call, or bus error, causes the monitor to emulate the effect of the trap on the currently scheduled virtual processor. After the currently scheduled VM's time slice is up, the virtual processor information is saved and the next virtual processor is swapped into the real processor.

2. Virtual Physical Memory

To virtualize physical memory, an extra level of address translation is added. This level maintains VM physical-to-machine address mappings. Virtual machines are given physical addresses that start at address zero and continue to the size of the VM's memory. These physical addresses could be mapped to machine addresses used by the Intel processor using the hardware-reloaded TLB of the Intel processor. The VMM manages the page table and does not allow the VM to insert entries in it. When the VMOS tries to insert a virtual-to-physical mapping in the TLB, the VMM emulates this by translating the physical address into the corresponding machine address and inserting this into the TLB.

3. Virtual I/O Devices

The VMM must intercept device accesses from virtual machines and forward them to physical devices. Instead of trying to use every device's real device driver, it is easier to use one special device driver for each type of device. Each device has a monitor call that is used to pass all command arguments to the VMM in a single trap. Many devices such as disks and network interfaces require direct memory access (DMA) to physical memory. Normally these device drivers use parameters that include a DMA

map. The VMM must intercept these DMA requests and translate physical addresses into machine addresses.

4. Virtual Network Interface

In order for VM's to communicate with each other, they use standard distributed protocols such as NFS. Disco manages a virtual subnet that allows this communication. The virtual subnet and networking interface used a copy-on-write strategy for transferring data between VMs to reduce the amount of copying. Virtual devices use ethernet addresses and do not limit the maximum transfer unit of packets, resulting in much faster communication.

E. PROBLEMS WITH CURRENT INTEL VMMS AND MANDATORY SECURITY ENFORCEMENT

If a computer system is to be a high-assurance secure computing system, it must be able to enforce security policies correctly, even under hostile attack. Examples of this type of system are those that are at least Class B2 or an equivalent level in the Common Criteria [Ref. 23]. Additionally, the systems' protection mechanisms must be structured and well-defined. In both an open and closed environment, the Yellow Book [Ref. 9] states that a system must be B1 or higher in order some users that access a system are not authorized for all categories. Even if Yellow Book requirements are not followed, two requirements should be met when dealing with classified information. First, labels are required when dealing with classified information. Second, for environments with multiple user clearances, a very effective protection mechanism is needed.

Current VMMs for the Intel architecture, such as VMware and FreeMWare, do not meet these requirements and they have never been evaluated as secure products (although a product does not have to be evaluated to be secure). Even though neither is designed to be secure, VMware does claim that their product can "isolate and protect each operating environment, and the applications and data that are running in it" [Ref. 7]. They also say that they do "not make any assumptions concerning the software that runs within the virtual machine. Even a rogue application or operating system is confined to

the VMware Virtual Platform." Given these claims, it is worthwhile to determine how well current VMMs can enforce isolation between VMs to support a mandatory security policy. It should be noted that the following analysis is based on assumptions of how VMware is accomplishing virtualization. The following sections describe some of the potential problems that would arise if VMware was to be used to separate mandatory security levels.

1. Resource Sharing

One of the simplest problems results from resource sharing between virtual machines. If two virtual machines have access to a floppy drive, information can flow from one VM to the other. This could be accomplished by simply copying files from the VM to the floppy so that other VM's could also access the files.

2. Networking and File Sharing

Another similar problem results from VMware's support of networking and file sharing. Using this support, two virtual machines at different security levels could communicate information to each other in a variety of ways. Some of these include Microsoft Networking, Samba, Novell Netware, Network File System, and TCP/IP. For example, using TCP/IP, a VM could FTP to either a host OS or guest Linux OS and transfer files.

3. Virtual Disks

VMware's ability to use virtual disks is also a problem. A virtual disk is a single file that is created in the host OS and used to encapsulate an entire disk, including an operating system and its applications. With this ability, anyone with access to this file in the host operating system could copy all of the information in this virtual disk to any sufficiently large type of external media. Once an attacker had the file, he could install VMware on his own operating system and open the virtual disk that contains all of the information.

Another problem with the virtual disk is that any OS application with read access to the file that contains the virtual disk in the host OS can examine the contents of all of the information on that virtual disk. For example, using Linux as the host operating

system, file utilities such as grep can be used to search for a specific string in the virtual file system. To test this vulnerability, Windows NT was booted as a VMware-supported guest OS on a Linux host OS. In NT, a text file with a known string such as "Top Secret Data" was created. Using the Linux host operating system, the *grep* command was used to search for this string in the approximately 300 MB virtual disk. The search succeeded in seconds. A solution to both this and the previous problem would be to not allow unauthorized users access to this virtual file. However, this would require a secure host OS.

4. Program Utilities

There is an alternative way to access data in a VMware virtual disk. VMware provides a utility called *vmware-mount* for a Linux host operating system. This utility allows virtual disks to be mounted as if they were regular hard disk partitions. Thus, if a user has permission to read a virtual hard disk file, the user can mount the disk and read file that it contains. The current version of this utility does not allow a virtual hard disk to be read and written at the same time. This means that this utility would only work if the virtual disk it is trying to access is not currently being used by a virtual machine.

One of the biggest security issues in VMware comes from another tool that it provides called VMware-Tools. VMware-Tools are available for each different type of guest operating system. These tools provide several nice features to support interoperation between virtual machines. Any user with the ability to run VMware can install the VMware tools in their virtual environment. For example, after installing VMware-Tools in a guest OS, the cursor can move freely between the host OS desktop and those of the VMs that are running as applications under of it. Another feature that these tools provide is the ability to cut and paste between virtual machines using something similar to the Windows clipboard. Text from any type of virtual machine can be copied and pasted into another virtual machine. It is not difficult to see the potential security danger if virtual machines were running at different mandatory security levels and information could be passed between windows.

5. Host Operating System

A host of security vulnerabilities emerge since both of the potential host operating systems for VMware (Linux and Windows NT) are weak platforms. Both of these operating systems have many of the features of a Class C2 operating system. However, the current versions of both of these operating systems have never been evaluated as such. Additionally, Class C2 systems do not enforce MAC policies. Since the security of VMware and its guest operating systems are a layer above the underlying operating system, it can not be expected to be any more secure than the least secure layer it depends on.

A security vulnerability in the VMware interface was announced on June 22nd, 1999. VMware was notified of a security problem in the first release: VMware for Linux version 1.0.1 (and all previous releases). The security hole allows a buffer overrun attack to result in unprivileged root access to the Linux host operating system. To exploit the security hole, a user must start VMware. Although the security alert does not describe how to exploit the hole, it does say that it must be done before powering on a virtual machine. Multi-user Linux systems with VMware installed would be affected the most by this attack. However, since this attack captures the whole system, no guest OS is safe either. A user of the system could execute the attack to gain root access on the machine. Once root is obtained, the user would be free to do anything he pleases.

6. Serial and Printer Ports

Another security problem in VMware occurs in the implementation of serial and printer ports. Before starting up a virtual machine, a configuration of the guest OS must be loaded or created. One of the options for parallel and serial ports in this configuration is for output of all parallel/serial ports to go to a file in the host operating system. If a printer is configured in the guest OS, the output will be sent to a file in the guest OS when the user tries to print something. This could be an easy way for users to transfer information from a high security class into the host OS. Anyone on the system could read the printer file in the host OS if its permissions were public.

F. A BETTER APPROACH FOR USING AN INTEL VMM TO SEPARATE MANDATORY SECURITY LEVELS

As the previous section demonstrated, current VMMs for the Intel architecture should not be used to separate mandatory security levels. Furthermore, it would not be wise to try to implement a high assurance virtual machine monitor (greater than B2) as a Type II VMM. This is because a Type II VMM must still operate in a host operating system. This means that a new highly secure host operating system for the Intel Pentium architecture would also have to be written. Layering a highly secure VMM on top of an operating system that is clearly not penetration resistant and the security controls of which can be circumvented would not provide a high level of security. This is because flaws in the underlying operating system could likely be exploited that would allow the security in the Class B2 or greater VMM to be bypassed.

A better approach to designing a VMM for the Intel architecture would be to build a Type I VMM as a microkernel. The Type I VMM would provide virtual environments on the machine. The VMM would intercept all attempts to handle low-level hardware functions from the VMs. Thus, the VMM would control all of the devices and system features of the CPU. The VMM would detect these attempts to use low-level hardware functions and transparently emulate their behavior for the virtual machine. The microkernel could allow each VM to choose among a specific set of devices. The devices may or may not be the real set of devices that are installed on the system.

There are two advantages to using a Type I VMM to separate mandatory security levels. First, a Type I VMM can provide a high degree of isolation between VMs. Second, all existing operating systems for the processor and their applications can be run in this highly secure environment without modification.

The biggest disadvantage to a Type I approach is that device drivers must be written for every device that needs to be used by the system. This is a problem because there are many different types of peripherals that can be hooked up to a system. Furthermore, each peripheral has many different manufacturers and each manufacturer's

product requires different drivers. A Type II VMM does not have this problem because it simply uses the drivers that are already written for the host OS. This disadvantage can be overcome when developing a secure solution by only supporting certain types and manufacturers of devices. For example, VMM designers may decide that they do not want the ability to hook up parallel disk drives to the system. Therefore, no drivers for parallel port drives need to be written. Additionally, drivers would not have to be written for all manufacturers of a particular device. For example, the designers could restrict supported printers to a particular type from a particular manufacturer. It is not out of the ordinary for highly secure solutions to require specific types of hardware.

If the secure VMM was implemented as a Type I VMM, the secure microkernel could be very small. This would make it easier for the VMM to meet the Reference Monitor verifiability requirement. For example, the VAX security kernel was only about 50,000 lines of code. Sufficient attention to constraining information flow between VMs combined with engineering inspection and testing techniques sufficient to demonstrate the absence of trap doors in the VMM would result in a secure VMM.

A tremendous advantage to using the VMM approach to separate mandatory security levels is that popular commercial operating systems and applications could be supported. It may be difficult to convince commercial software companies to port their software to a new platform if the market will be small. A VMM eliminates the need to port software and allows all of the functionality that computer users are used to having on their own desktop. This saves a tremendous amount of development and cost.

Before trying to implement a secure Type I VMM for the Intel architecture, it might be advantageous to convince a chip manufacturer to slightly modify the Intel Pentium architecture for the project. Two alternative modifications to the processor could make virtualization easier. First, all seventeen unprivileged, sensitive instructions of the Intel architecture could be changed into privileged instructions. This would make virtualization easier because the VMM would no longer have to look ahead for possible sensitive instructions. All instructions would naturally be able to trap so that the VMM

could emulate the behavior of the instruction. However, this solution may cause problems in current COTS operating systems because instructions that were non-privileged will now trap when they did not previously.

The second alternative is to implement a trap on op-code instruction. [Ref. 1] This solution adds a new instruction to the architecture that allows an operating system to declare instructions that should be treated as if they were privileged. This alternative makes virtualization easier without affecting current COTS operating systems.

If one of the alternatives described above is not implemented, virtualizing the Intel architecture is much harder. This is because additional code is required to force sensitive, unprivileged instructions to trap so that they can be handled by VMM software. The additional code raises two security concerns. First, the security kernel may not be considered minimal because of the extra virtualization code. Second, virtualization of the unmodified processor requires checking every instruction before it executes to determine if it is one of seventeen problem instructions. If undocumented instructions exist, the VMM may not operate as expected because it will not recognize an undocumented instruction. Furthermore, if an undocumented instruction, or sequence of instructions, exists that cause the processor to transition into an insecure state, an attacker may be able to bypass the VMM.[6]

Modifying the problem instructions of the Intel architecture or adding a trap on op-code instruction will make a Type I VMM more secure and easier to build. The Intel architecture has many features that can be used to implement highly secure systems. The best way to implement a secure VMM on the Intel architecture would be to build a new, high assurance Type I VMM on a slightly modified processor. Even though device drivers

---

6 To ensure that a completely secure solution can be built on the Intel architecture, it is necessary to have some sort of guarantee that these instructions or sequences of instructions do not exist. It has already been discovered that Intel has undocumented instructions in their architecture. Dr. Dobbs' Journal [Ref. 24] illustrates several undocumented instructions including UMOV (User Move Data), LOADALL (loads the entire CPU state), ICEBP (ICE Breakpoint), and SALC (Set AL on Carry).

will have to be written, a Type I VMM will be more secure and easier to verify than a Type II VMM implementation on a low assurance operating system.

VI. CONCLUSION

This thesis explored the feasibility of implementing a secure virtual machine monitor on the Intel Pentium architecture. The thesis began by covering the types of virtual machine monitors and their hardware requirements as described by Goldberg. Then, a detailed study of the Intel architecture was done to see if it could meet the hardware requirements of any type of VMM. A large part of this study involved looking at approximately 250 instructions in the Intel architecture to determine if they are virtualizable. The analysis showed that seventeen instructions did not meet Goldberg's requirements because they were sensitive and unprivileged.

Even though the Intel architecture is not truly virtualizable, a product exists that provides something similar to a Type II virtual machine for the Intel architecture. This class of VMM was examined to determine how a VMM can run on an architecture that is not virtualizable. The analysis showed that this effect may be accomplished by examining all instructions before they are executed. If a problem instruction is found, the VMM patches the instruction with code that will force a trap to the VMM. The VMM is then able to emulate the proper behavior of the instruction.

After defining a strategy that can be used to "virtualize" the Intel architecture, an analysis was conducted to determine whether a secure virtual machine monitor could be built for the architecture to separate classified from unclassified information. The VAX Security Kernel was described to give an example of a secure virtual machine monitor. We conclude that current VMM products for the Intel architecture should not be used as a secure virtual machine monitor, even though some vendors claim security as a feature [Ref. 25]. One of the VMM products claims that virtual machines are isolated in a way that prevents them from affecting each other or the host operating system. However, since the product resides on top of an operating system that is not secure, it can not provide a high enough degree of isolation to protect information in a multilevel environment.

The best way to separate mandatory security classes when supporting virtual machines would be to build a new, secure Type I VMM on the Intel architecture. The Intel architecture has features that can be used to implement highly secure systems. There are many advantages to building a highly secure VMM on the Intel architecture. First and most important, this solution allows high security while providing a platform that is compatible with existing popular COTS operating systems and applications.

A. FUTURE WORK

Three additional areas of research could supplement the work contained in this thesis. First, one might design a secure Type I VMM for the Intel Pentium architecture. The design should consider the "pitfalls" that were mentioned in Chapter V of this thesis.

Second, future work could include analyzing how the source code from the FreeMWare project (when it is available) could be changed to implement a secure Type I VMM. Even though the FreeMWare project is a Type II VMM, a significant amount of the code could probably be used to begin the effort.

Third, it would be useful to look at the Intel IA64 architecture and determine how its new features help or hurt virtualization. It is rumored that the architecture will have five rings, allowing any Intel processor from the 8086 to the Pentium III to be virtualized in its four outer rings.

APPENDIX A. INTEL PENTIUM III ARCHITECTURE REVIEW

A. ARCHITECTURE

In order to discuss whether the Intel Pentium processor is capable of supporting a secure VMM, I must give a brief overview of the Pentium architecture. For more detailed information, see the Intel manuals [Ref. 6, 26, 27].

Figure 9 illustrates the Pentium Pro processor architecture. The architecture has five subsystems: memory, fetch/decode unit, instruction pool, dispatch/execute unit, and the retire unit.

B. MEMORY MODELS

The address space for physical memory is organized as a long sequence of 8-bit bytes, each having a unique address. When programs use the processor's memory management capabilities, they do not directly address physical memory. Instead, they use one of three memory models illustrated in Figure 10.

In the flat memory model, memory is a single, contiguous address space. The code, data, and procedure stacks are all contained within this space and are byte-addressable. In the segmented memory model, memory is a group of independent address spaces called segments. Code, data, and stacks are contained in separate segments that are up to 2^{32} bytes in size. A logical address, consisting of a segment and an offset, is used to address a byte in a segment. The segmented memory model increases the reliability of programs and systems by separating code, data, and stack segments. For example, it prevents a stack segment from being able to "grow" into code or data space and thus overwrite instructions or data. Furthermore, separating OS code, data, and stack segments from application segments protects OS segments from application segments and vice versa. The third memory model is the real-address model which was used in the 8086 processor. It is a specific implementation of the segmented memory model where segments are only 2^{16} bytes (64K) each.

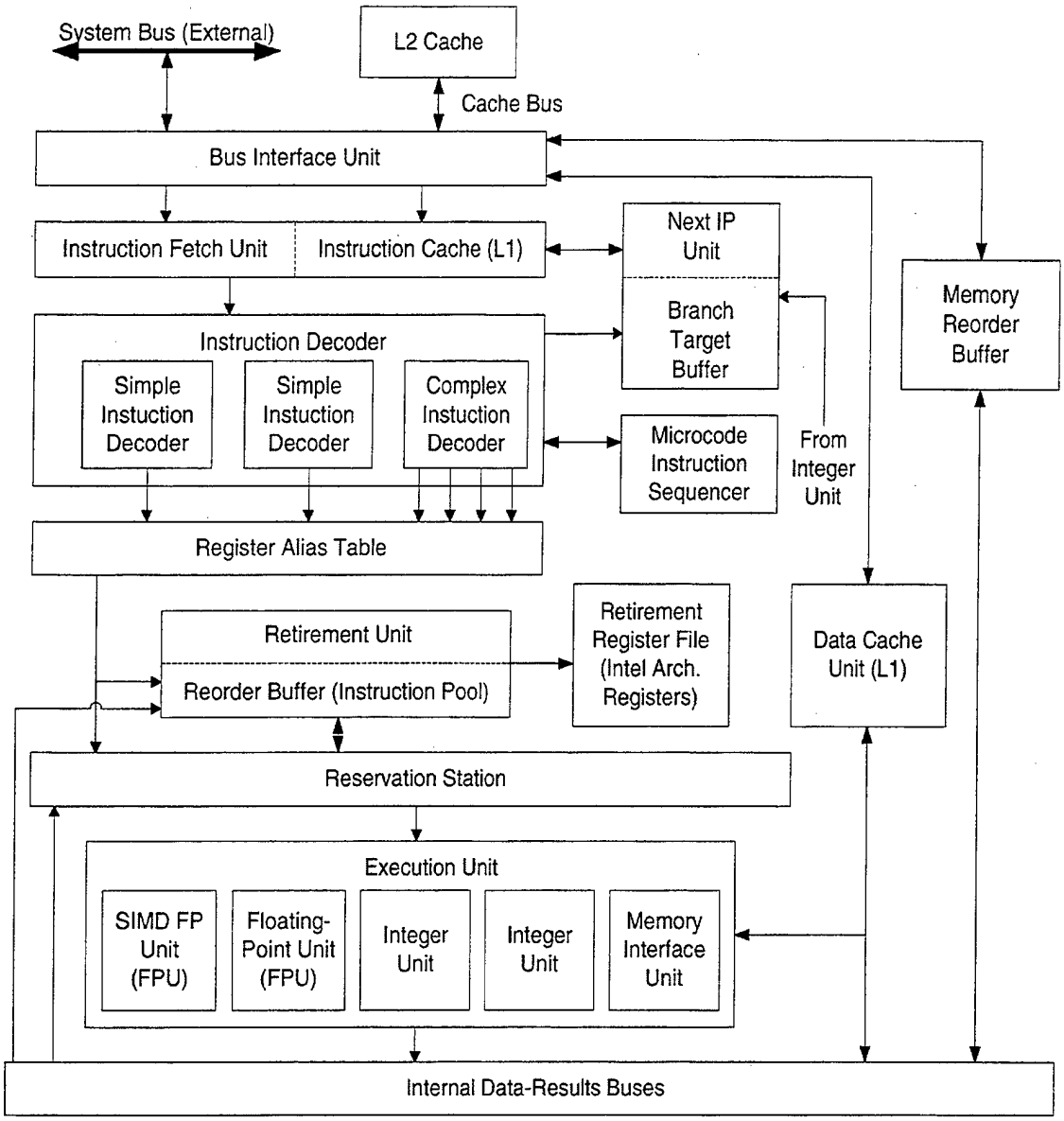


Figure 9. Pentium Pro Processor Microarchitecture From Ref. [26].

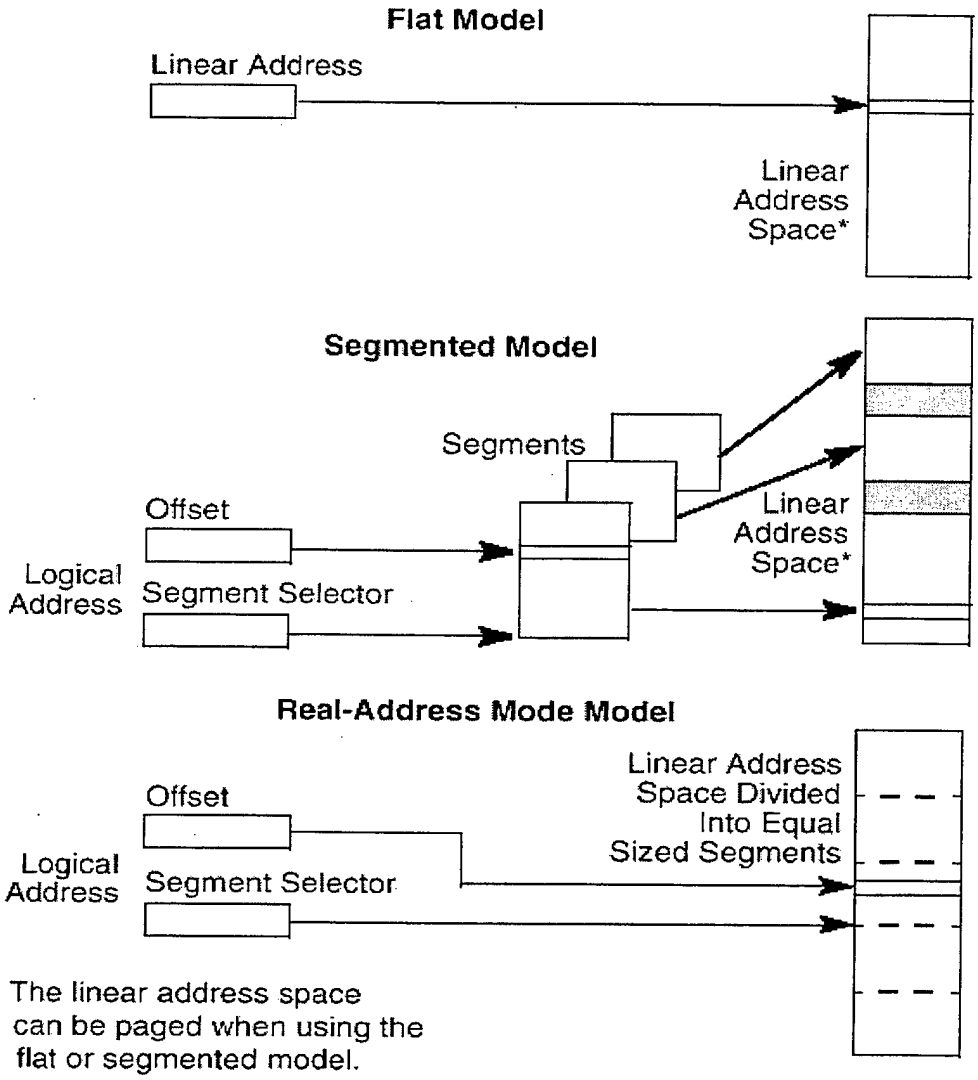


Figure 10. Three Memory Management Models From Ref. [26].

C. EXECUTION ENVIRONMENT

All programs or tasks running on the Intel architecture have a set of resources to execute instructions and store code, data, and state. These resources include are in the table below.

Resource	Size	Quantity	Name(s)
General-Purpose Registers	32-bit	8	EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP (E = extended)
Segment Registers	16-bit	6	CS, DS, SS, ES, FS, GS
EFLAGS Register	32-bit	1	EFLAGS
EIP (Instruction Pointer Register)	32-bit	1	EIP
Address Space	$0\text{-}2^{32}$ bytes	1	N/A

Table 4. Intel Program Resources.

The Intel Pentium's execution environment has three operating modes. First, protected mode has all of the instructions and architectural features of the processor available to it. This mode has the most capability of the three and can use any of the three memory models described in the previous section. To execute 8086 software, protected mode is used to create a "virtual-8086" mode. This mode is not considered one of the three operating modes of the processor, but allows 8086 software to run in a protected, multi-tasking environment.

The second processor mode is the real-address mode. This mode provides the programming environment of the 8086 processor with a few added capabilities. In this mode, the processor only supports the real-address memory model. The major difference between real-address mode and virtual-8086 mode is that the virtual-8086 mode uses some protected mode services such as protected mode interrupt handling, exception handling, and paging.

Finally, the third mode is system management mode. This mode provides an operating system a transparent mechanism to implement functions such as power management and system security. In this mode, the processor switches to a separate address space called the system management RAM and saves the context of the current program or task. System management mode uses a memory model similar to the real-address memory model.

System management mode (SMM) is designed to handle system-wide functions such as power management, system hardware control, or proprietary OEM-designed

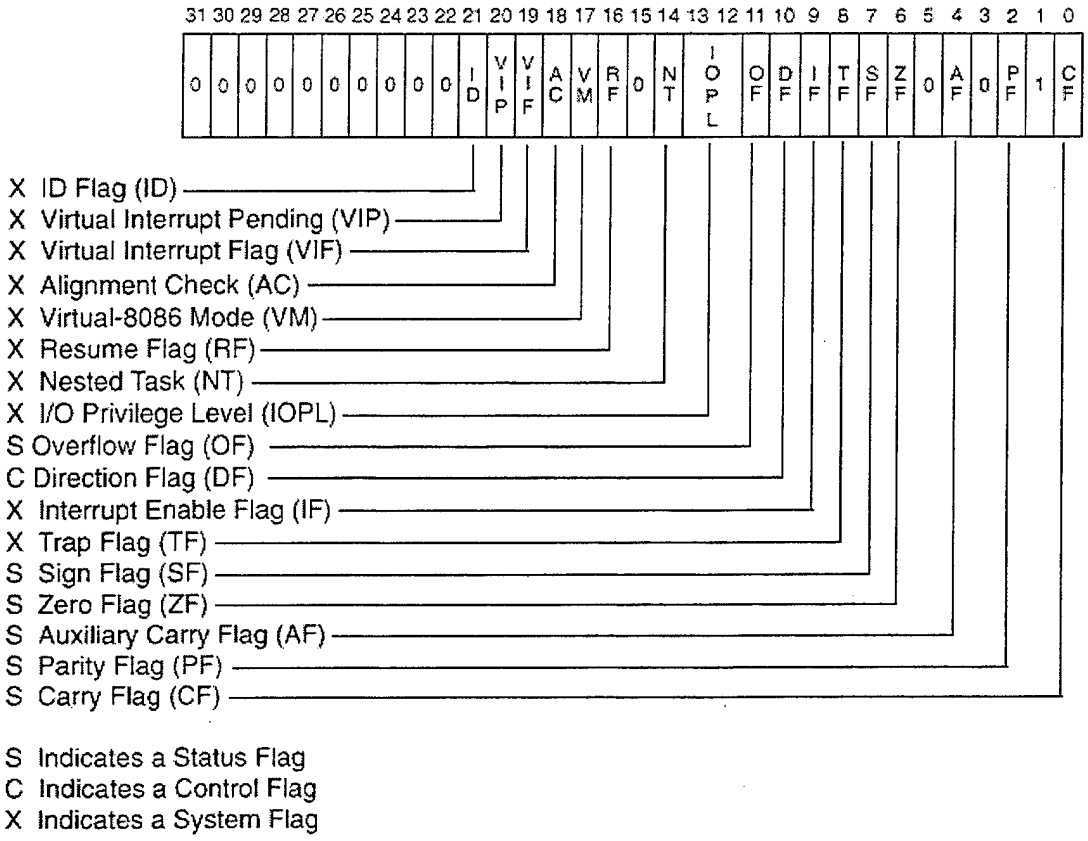
code. It is intended for use only by system firmware, not by application software or system software. SMM offers a distinct, isolated processor environment that operates transparently to an OS and its applications. To start system management mode, a system management interrupt (SMI) is signaled. The SMI is a nonmaskable, external interrupt that is unlike the processor's normal interrupt and exception mechanism. After invoking the system management interrupt, the processor saves its current state and switches to a separate operating environment in system management RAM. It then executes system management interrupt handler code before switching back to the previous processor state. The system management mode does not use privilege levels or address mapping and is capable of addressing up to 4 gigabytes of memory. It can also execute all I/O and system instructions.

D. EFLAGS REGISTER

The Intel architecture has a 32-bit EFLAGS register that contains status flags, system flags, and a control flag. The EFLAGS register is illustrated in Figure 11 below. The status flags show the results of arithmetic operations. The system flags and I/O privilege level flag control the operations of the operating system. The operating system behaves differently based on how these flags are set. System flags and the IOPL should not be modified by application programs.

E. PROTECTION MECHANISM

The Intel architecture uses a protection mechanism that implements four different privilege levels: 0, 1, 2, and 3 (0 is the highest privilege level). The mechanism provides the ability to limit access to segments and pages based on privilege level. Privilege levels 0, 1, and 2 are considered supervisor mode and privilege level three is considered user mode. The page-level protection mechanism determines access using two different privilege levels: supervisor and user. Using the protection mechanism, all memory references are checked before allowing access. These checks include limit, type,



privilege level, restriction of addressable domain, restriction of procedure entry-points, and restriction of instruction set.

Figure 11. The EFLAGS Register From Ref. [26].

In protected mode, the processor has three types of privilege levels. The current privilege level (CPL) is the privilege level of the current executing task. The descriptor privilege level (DPL) is the privilege level of a segment or gate. Finally, the requested privilege level (RPL) is an "override" privilege level that is assigned to segment selectors.

To allow access to code segments with different privilege levels, the processor uses gate descriptors. Code modules can only access modules in higher privilege segments by using a protected interface called a gate. Trying to perform a transfer to such a segment without using a gate causes a general-protection exception. When a call

to a procedure that is in a more privileged protection level than the procedure that is calling it is made, the following steps occur:

First, the segment selector in the CALL instruction references a call gate descriptor. The call gate descriptor provides access rights information, the segment selector for the code segment of the called procedure, and the offset into the code segment (the instruction pointer). Four privilege levels are used to check the validity of a program control transfer using a call gate. They are the CPL, RPL (of the gate's selector), DPL (of the call gate descriptor), and the DPL (of the segment descriptor of the destination code segment. Based on whether control transfer was initiated with a CALL or JMP instruction, privilege checking rules are as follows:

Instruction	**Privilege Check Rules**
CALL	CPL <= call gate DPL; RPL <= call gate DPL
	Destination conforming code segment DPL <= CPL
	Destination nonconforming code segment DPL <= CPL
JMP	CPL <= call gate DPL; RPL <= call gate DPL
	Destination conforming code segment DPL <= CPL
	Destination nonconforming code segment DPL = CPL

Table 5. Privilege Checking Rules for Call Gates After Ref. [27].

Second, the processor switches to a new stack to execute the called procedure because each privilege level has its own stack. Therefore, each task must have a stack for its own privilege level and a stack for each privilege level higher than its own. Privilege level 3 uses the SS segment selector and the ESP as a stack pointer. The segment selectors and stack pointers for the other more privileged levels are stored in a system segment called the task state segment.

F. INTERRUPTS AND EXCEPTIONS

Interrupts and exceptions are a way to force a transfer of execution from the currently running program or task to a special procedure or task called a handler. When an interrupt or exception is detected, the current task is automatically suspended while the processor executes the handler. After the handler is executed, the interrupted task resumes without loss of program continuity. The Intel architecture has 16 predefined interrupts and exceptions and 224 user-defined, or maskable, interrupts. All interrupts and exceptions have associated entries in the interrupt descriptor table, or IDT. The IDT contains a collection of gate descriptors (interrupt, trap, or task). Each interrupt or exception in the IDT is identified by a number called a vector. When the processor detects an interrupt or exception it executes an implicit call to a handler procedure or handler task.

The processor can receive an interrupt from two sources: an external, hardware-generated interrupt or a software-generated interrupt. External interrupts are delivered using a special pin on the processor or through an APIC (Advanced Programmable Interrupt Controller). Software generated interrupts are created by using the INT instruction and by supplying the interrupt vector number as an operand. Any of the 256 interrupt vectors can be used as a parameter to the INT instruction.

Exceptions are divided into three classes: faults, traps, and aborts. They are divided based on the way they are reported and whether the instruction that caused the exception can be restarted with no loss of program continuity. A fault can be corrected and allows a program to resume with no loss of continuity. When a fault is reported, the processor restores the machine state to the state prior to execution of the faulting instruction. A trap is reported immediately after a trapping instruction, allowing the execution of a program to be resumed without loss of program continuity. An abort does not always report the precise location of the instruction that caused the exception and does not allow restarting of the task that caused the exception.

The processor can receive an exception from three sources: processor detected program error exceptions, software-generated exceptions, or machine check exceptions. Program error exceptions are generated by the processor if it detects an error while executing an application program. Three instructions-- INTO, INT 3, and BOUND-- allow exceptions to be generated by software. Finally, machine-check exceptions are generated by a mechanism in the processor that checks for processor errors.

To access an interrupt or exception handler, internal hardware or software must send an interrupt vector to the processor. The vector indexes into the IDT (Interrupt Descriptor Table) to a gate descriptor for the procedure or task used to service the interrupt or exception. The IDT can contain three kinds of descriptors: task-gate, interrupt-gate, and trap-gate descriptors. If the descriptor is a task gate, the handler is accessed using a task switch. If the descriptor is an interrupt or trap gate, the handler is accessed using a method similar to a call gate. The IDT can reside anywhere in the linear address space and is located using the IDT register.

When the processor performs a call to an interrupt or exception handler, it saves the current state of the EFLAGS register, CS register, and EIP register on the stack. Upon returning to the calling task, the IRET instruction is used to restore the saved flags into the EFLAGS register. However, the IOPL field is restored only if the CPL is 0. Additionally, the IF flag is changed only if the CPL is less than or equal to the IOPL.

G. INPUT/OUTPUT

Intel processors can send data to and get data from input/output ports (I/O ports). System hardware creates I/O ports that are configured to communicate with peripheral devices. There are three types of I/O ports: input, output, and bi-directional.

I/O ports can be accessed two ways: with a separate I/O address space or with memory-mapped I/O (in physical memory). Accessing I/O through the former method is done using a set of I/O instructions and an I/O protection mechanism, all of which are provided by the processor. Accessing I/O with the latter method is done by using the

processors general-purpose move and string instructions. In this case, protection is provided through segmentation and/or paging. I/O ports can be mapped to appear in the I/O address space, the physical memory address space, or both.

The processor's I/O address space is separate from the physical memory address space. The I/O address space consists of 64K individually addressable 8-bit I/O ports. These ports are numbered from 0 to FFFFH. There are two protection devices that regulate access to I/O ports. The first is the I/O privilege level (IOPL field of the EFLAGS register) and the second is the I/O permission bit map of a task state segment.

When I/O devices use memory-mapped I/O, any of the processor's instructions that reference memory can be used to access an I/O port at a certain physical address. This means that normal segmentation and paging mechanisms apply.

H. ADDITIONAL SYSTEM REGISTERS

The Intel Pentium system architecture supports many system operations such as memory management, protection of software modules, task management, control of multiple processors, interrupt and exception handling, cache management, hardware and resource power management, and debugging and performance monitoring.

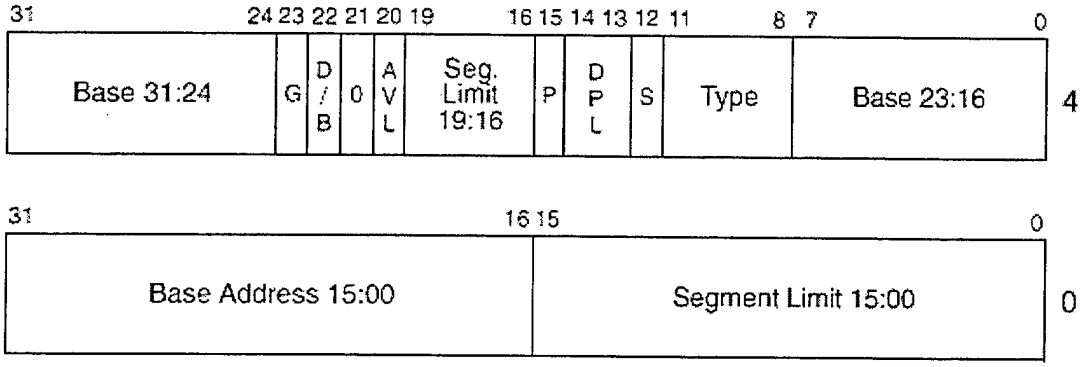
The system architecture uses many registers which have not yet been discussed. These include control registers (CR0, CR1, CR2, CR3) for system level operations, debug registers for debugging programs, the GDTR, LDTR, IDTR registers, and the task register. The control registers are illustrated in Figure 12 below. The GDTR, LDTR, IDTR, and task register all contain the linear address and size of their respective table or task. The system level registers and data structures are illustrated in Figure 13 below.

I. TASK MANAGEMENT

In protected mode, all processor execution is performed with tasks. A task is a "unit of work that a processor can dispatch, execute, and suspend. It can be used to execute a program, a process, an operating-system service utility, an interrupt or

exception handler, or a kernel or executive utility" [Ref. 27]. The Intel architecture has mechanisms that allow tasks to be saved, executed, and switched.

A task consists of two parts. The first is a task execution space that has a code segment, stack segment, and one or more data segments. If protection mechanisms are used, the task execution space also contains a stack for each privilege level higher than its



AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level
G — Granularity
LIMIT — Segment Limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type

Figure 12. Control Registers From Ref. [27].

own. The second part is the task state segment, or TSS. A TSS defines a task's execution environment state. The state of the executing task is defined by the following [Ref. 27].

- The task's current execution space, defined by the segment selectors in the segment registers (CS, DS, SS, ES, FS, and GS).
- The state of the general-purpose registers.
- The state of the EFLAGS register.
- The state of the EIP register.

- The state of control register CR3.
- The state of the task register.
- The state of the LDTR register.
- The I/O map base address and I/O map (contained in the TSS).
- Stack pointers to privilege 0, 1, and 2 stacks (contained in the TSS).
- Link to previously executed task (contained in the TSS).

All of these items other than the state of the task state register are contained in the task's TSS before it is dispatched.

The TSS identifies the segments that make up the task execution space and has a storage space for task state information. A task is identified using a segment selector to its TSS. The task register holds the TSS for the current task. If paging is implemented, the base address of the task's page directory is loaded in control register 3.

There are five ways to execute a task [Ref. 27]:

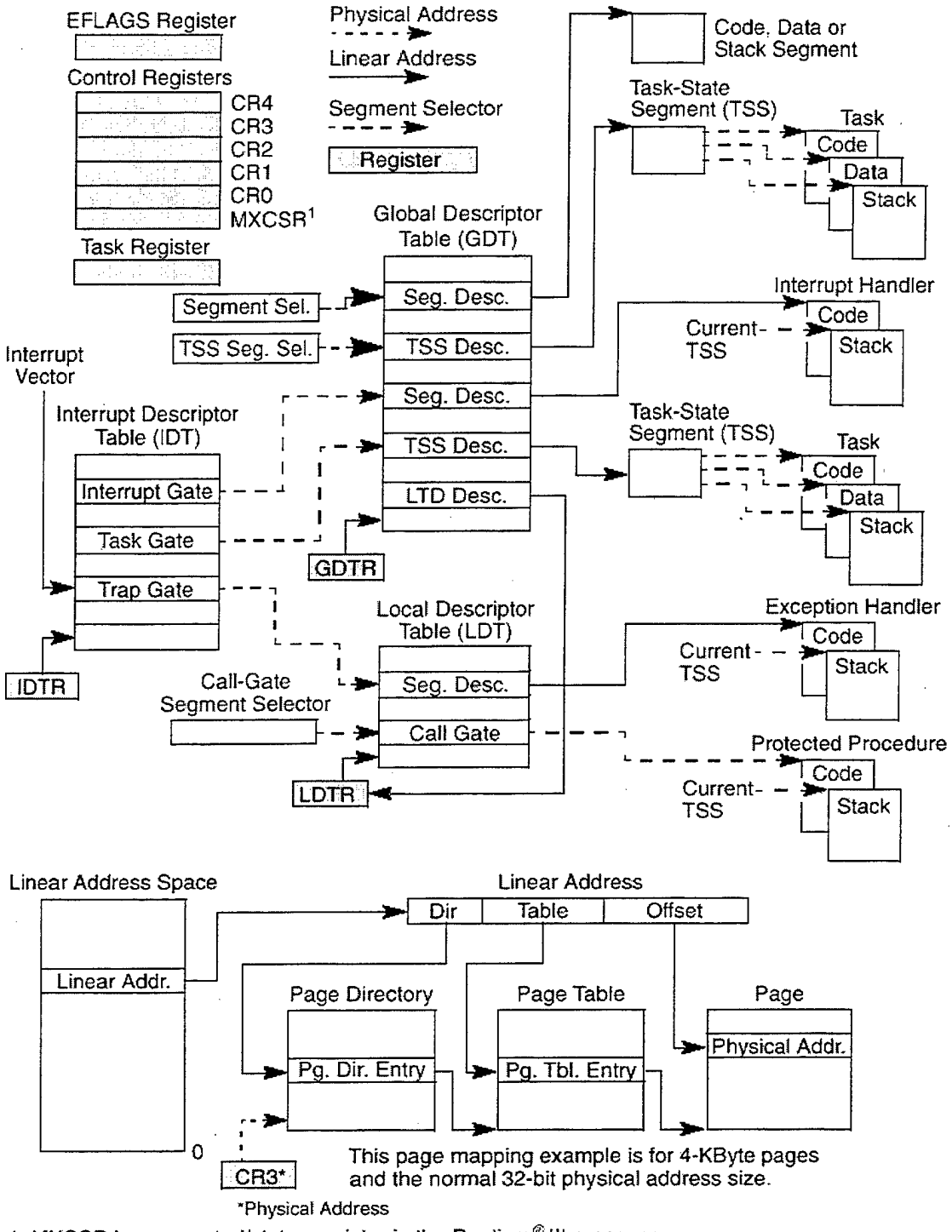
- An explicit call to a task with the CALL instruction.
- An explicit jump to a task with the JMP instruction.
- An implicit call (by the processor) to an interrupt-handler task.
- An implicit call to an exception-handler task.
- A return (initiated with an IRET instruction) when the NT flag in the EFLAGS register is set.

There are four ways to execute a task switch [Ref. 27]:

- The current program, task, or procedure executes a JMP or CALL instruction to a TSS descriptor in the GDT.
- The current program, task, or procedure executes a JMP or CALL instruction to a task-gate descriptor in the GDT or the current LDT.

• An interrupt or exception vector points to a task-gate descriptor in the IDT.

• The current task executes an IRET when the NT flag in the EFLAGS register is set.



1. MXCSR is new control/status register in the Pentium® III processor.

Figure 13. System Registers and Data Structures From Ref. [27].

J. PROCESSOR MANAGEMENT AND INITIALIZATION

After a machine is powered up or reset, every processor on the bus performs a hardware reset. Every processor sets its registers and floating point unit to a known state and enters real address mode. This state varies based on the processor family. If there is more than one processor on the system bus, a protocol runs to assign a primary processor. After all processors are initialized, configured, and synchronized, the primary processor begins executing an operating system or executive task. The first instruction that is executed is located at physical address FFFFFFF0H.

Since the processor starts in real address mode, the only data structure that has to be loaded into memory is the interrupt vector table. However, before the processor can switch to protected mode, the following data structures and code modules must be loaded into memory by the software initialization code [Ref. 27]:

- A protected-mode IDT.
- A GDT.
- A TSS.
- (Optional.) An LDT.
- If paging is to be used, at least one page directory and one page table.
- A code segment that contains the code to be executed when the processor switches to protected mode.
- One or more code modules that contain the necessary interrupt and exception handlers.

Furthermore, the software initialization code must also initialize the following registers: GDTR, IDTR, control registers CR1 through CR4, and memory type range registers (MTRRs). MTRRs are only applicable to the Pentium Pro processor and beyond. The MTRRs allow memory to be associated with physical-address ranges in

system memory. This allows the processor to optimize operations for different types of memory such as RAM, ROM, and memory-mapped I/O devices.

To enter protected mode, the PE bit of CR0 is set. Once in protected mode, software usually does not switch back to real mode. This is because it can run the code in virtual 8086 mode. However, clearing the PE bit in the EFLAGS register will bring the processor back into real address mode.

In protected mode, all memory accesses pass through the global descriptor table (GDT) or the local descriptor table (LDT). Both of these tables contain segment descriptors that contain the base address of a segment in linear address space, access rights, type, and usage information. Every segment descriptor has a segment selector that contains a global/local flag, access rights information, and an index into the GDT or LDT. To access a byte of memory, a segment selector and offset are supplied.

K. GATES

The system architecture contains a set of special descriptors called gates. There are four types of gates: call, interrupt, trap, and task. They provide protected gateways to system procedures/handlers that operate at a more privileged level than normal applications.

A call gate works as follows. First, a calling procedure provides the selector of a call gate. Then the processor performs a check on the access rights of the call gate. This is done by comparing the CPL (current privilege level) with the PL of the call gate and the destination code segment that the call gate points to. If access to the destination segment is allowed, the procedure gets a segment selector and an offset into the destination code segment from the call gate. If a change in privilege level is required, the processor also switches to the stack for that privilege level.

L. MEMORY MANAGEMENT

The system architecture supports two types of memory management facilities: segmentation and virtual memory using paging. These facilities use three types of memory addresses: logical addresses, linear addresses, and physical addresses.

The Intel architecture provides a physical address space of 4GB. The physical address space is the range of addresses the processor can address on its address bus. When paging is used, a linear address is mapped to the physical address space. All segments are contained in the linear address space. Paging is a mechanism to use a virtual memory system where sections of a program's execution environment are mapped into physical memory as they are needed.

When segmentation is used, logical addresses are mapped into the linear addresses space. Segmentation provides a mechanism to divide the linear address space into small, protected address spaces (segments). Segments are used to hold the code, data, and stack segments for programs and to hold system data structures such as a task state segment or a local descriptor table. The processor enforces the boundaries between these segments and does not allow one program to write into another program's segments. All segments are contained in the processor's linear address space. A logical address is mapped to a linear address using a segment selector and an offset. The base address of the segment selector and the offset together form a linear address.

If the processor is in protected mode (the normal operating mode of the processor), it is not possible to disable segmentation. The use of paging however is optional. If paging is not used, linear addresses are mapped directly into the physical address space of the processor. When using paging, each segment is divided into pages that are typically 4 kilobytes in size. Every linear address is broken into three parts. These parts provide offsets into the page directory, page table, and the page frame. A page directory entry contains the physical address of the base of a page table, access rights, and memory management information. A page table entry contains the physical address of a page frame, access rights, and memory management information. When a

program attempts to use a linear address, the processor uses the page directory and page tables to translate the linear address into a physical address and performs the requested operation on the memory location. If the page is not in physical memory, the processor interrupts execution of the program with a page fault exception and reads the page into physical memory from the disk and continues executing the program (see Figure 14 below). The processor stores the most recently used page-directory and page-table entries in a cache on the processor called translation lookaside buffers (TLBs).

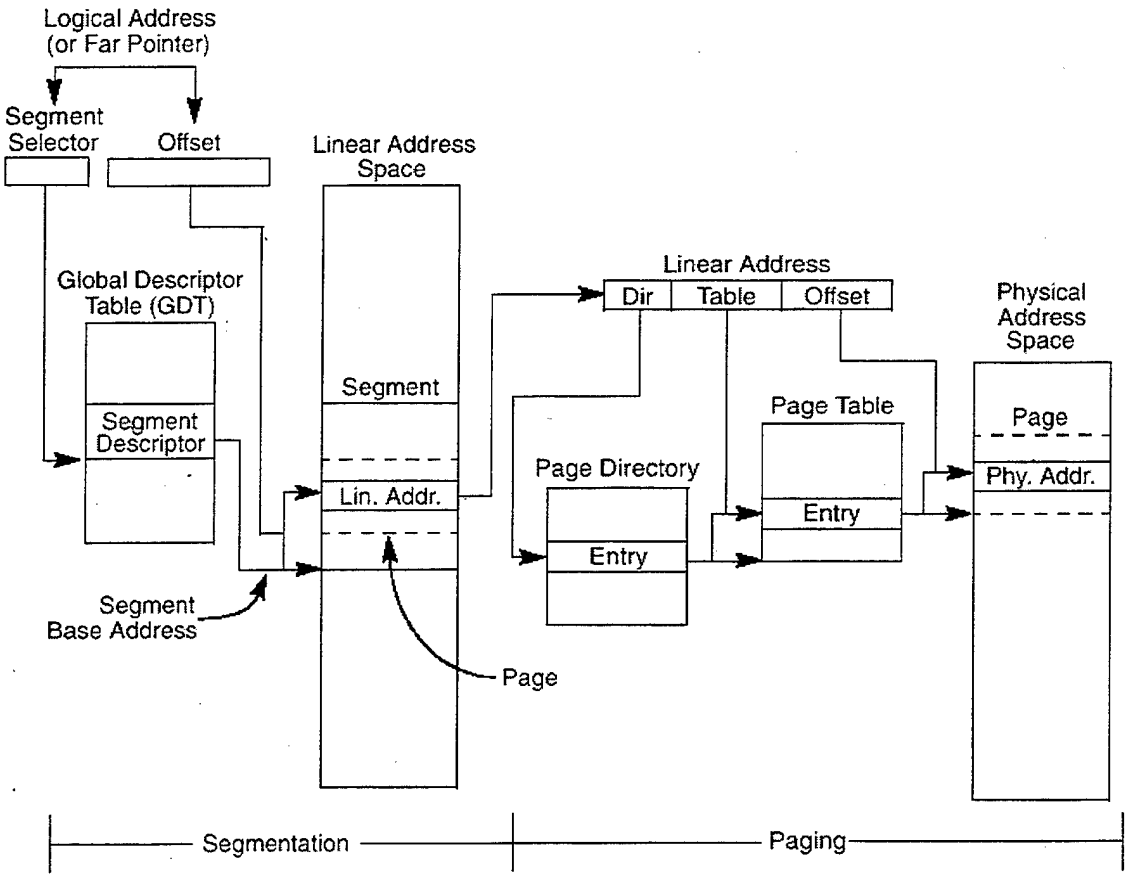


Figure 14. Segmentation and Paging From Ref. [27].

APPENDIX B. INTEL INSTRUCTIONS

This appendix contains a list of Intel instructions, their class, and whether or not they are sensitive, privileged, or problem instructions. The following is a key for abbreviations that are used in the instruction list.

NAME	The name of the instruction
DESCRIPTION	A short description of the instruction
SENS	Is the instruction sensitive?
PRIV	Is the instruction privileged?
PROB	Is the instruction a problem for virtualization?
CLASS	The Intel-defined class that the instruction falls under (defined below)

I1 = Integer Data Transfer	F1 = Floating Point Data Transfer
I2 = Integer Binary Arithmetic	F2 = Floating Point Basic Arithmetic
I3 = Integer Decimal Arithmetic	F3 = Floating Point Comparison
I4 = Integer Logic Instructions	F4 = Floating Point Transcendental
I5 = Integer Shift and Rotate	F5 = Floating Point Load Constraints
I6 = Integer Bit and Byte	F6 = Floating Point Unit Control
I7 = Integer Control Transfer	ST1 = Streaming SIMD Extensions Data Transfer
I8 = Integer String	ST2 = Streaming SIMD Extensions Conversion
I9 = Integer Flag Control	ST3 = Streaming SIMD Extensions Packed Arithmetic
I10 = Integer Segment Register	ST4 = Streaming SIMD Extensions Comparison
I11 = Integer Miscellaneous	ST5 = Streaming SIMD Extensions Logical
M1 = MMX Data Transfer	ST6 = Streaming SIMD Extensions Data Shuffle
M2 = MMX Conversion	ST7 = Streaming SIMD Extensions Additional SIMD-Integer
M3 = MMX Packed Arithmetic	ST8 = Streaming SIMD Extensions Cacheability Control
M4 = MMX Comparison	ST9 = Streaming SIMD Extensions State Management
M5 = MMX Logic	S = System
M6 = MMX Shift and Rotate	
M7 = MMX State Management	

REASON The reason the instruction is a problem for virtualization

3A1 = Mode of the VM	References or changes the mode of the processor: real-address, protected, system management mode, etc.
3A2 = State of the machine	References or changes the state of the processor: halt, respond to interrupts, respond to debug exceptions, etc.
3B1 = Sensitive Registers	References or changes sensitive registers: CR0, LDTR, EFLAGS, etc.
3B2 = Sensitive Memory Locations	References or changes sensitive memory locations
3C1 = Protection system	Reference or change a privilege level in the system: CPL, IOPL, etc.
3C2 = Memory System	Reference the memory system: alignment checking, invalidating cache and TLB entries, etc.
3C3 = Address Relocation System	Interferes with how physical, linear, and logical addresses are translated
3D = I/O Instructions	Move data between the processor's I/O ports and a register/memory

NAME	DESCRIPTION	SENS	PRIV	CLASS	REAS	PROB
AAA	ASCII Adjust After Addition	N	N	I3	-	N
AAD	ASCII Adjust AX Before Division	N	N	I3	-	N
AAM	ASCII Adjust AX After Multiply	N	N	I3	-	N
AAS	ASCII Adjust AL After Subtraction	N	N	I3	-	N
ADC	Add with Carry	N	N	I2	-	N
ADD	Add	N	N	I2	-	N
ADDPS	Packed Single-FP Add	N	N	ST3	-	N
ADDSS	Scalar Single-FP Add	N	N	ST3	-	N
AND	Logical AND	N	N	I4	-	N
ANDNPS	Bit-wise Logical And Not For Single-FP	N	N	ST5	-	N
ANDPS	Bit-wise Logical And For Single FP	N	N	ST5	-	N
ARPL	Adjust RPL Field of Segment Selector	N	N	S	-	N
BOUND	Check Array Index Against Bounds	N	N	I7	-	N
BSF	Bit Scan Forward	N	N	I6	-	N
BSR	Bit Scan Reverse	N	N	I6	-	N
BSWAP	Byte Swap	N	N	I1	-	N
BT	Bit Test	N	N	I6	-	N
BTC	Bit Test and Complement	N	N	I6	-	N
BTR	Bit Test and Reset	N	N	I6	-	N
BTS	Bit Test and Set	N	N	I6	-	N
CALL	Call Procedure	Y	N	I7	3C1	Y
CBW/CWDE	Convert Byte to Word/Convert Word to Doubleword	N	N	I1	-	N
CLC	Clear Carry Flag	N	N	I9	-	N
CLD	Clear Direction Flag	N	N	I9	-	N
CLI	Clear Interrupt Flag	Y	Y	I9	3C1	N
CLTS	Clear Task-Switched Flag in CR0	Y	Y	S	3B1	N
CMC	Complement Carry Flag	N	N	I9	-	N
CMOVcc	Conditional Move	N	N	I1	-	N
CMP	Compare Two Operands	N	N	I2	-	N
CMPPS	Packed Single-FP Compare	N	N	ST4		N
CMPS/CMPSB /CMPSW/CMP SD	Compare String Operands	N	N	I8	-	N
CMPSS	Scalar Single-FP Compare	N	N	ST4	-	N
CMPXCHG	Compare and Exchange	N	N	I1	-	N
CMPXCHG8B	Compare and Exchange 8 Bytes	N	N	I1	-	N
COMISS	Scalar Ordered Single-FP Compare and Set EFLAGS	N	N	ST4	-	N
CPUID	CPU Identification	N	N	I11	-	N
CVTPI2PS	Packed Signed INT32 to Packed Single-FP Conversion	N	N	ST2	-	N
CVTPS2PI	Packed Single-FP to Packed INT32 Conversion	N	N	ST2	-	N

CVTSI2SS	Scalar Signed INT32 to Single-FP Conversion	N	N	ST2	-	N
CVTSS2SI	Scalar Single-FP to Signed INT32 Conversion	N	N	ST2	-	N
CVTTPS2PI	Packed Single-FP to Packed INT32 Conversion (Truncate)	N	N	ST2	-	N
CVTTSS2SI	Scalar Single-FP to Signed INT32 Conversion (Truncate)	N	N	ST2	-	N
CWD/CDQ	Convert Word to Doubleword/Convert Doubleword to Quadword	N	N	I1	-	N
DAA	Decimal Adjust AL after Addition	N	N	I3	-	N
DAS	Decimal Adjust AL after Subtraction	N	N	I3	-	N
DEC	Decrement by 1	N	N	I2	-	N
DIV	Unsigned Divide	N	N	I2	-	N
DIVPS	Packed Single-FP Divide	N	N	ST3	-	N
DIVSS	Scalar Single-FP Divide	N	N	ST3	-	N
EMMS	Empty MMX™ State	N	N	M6	-	N
ENTER	Make Stack Frame for Procedure Parameters	N	N	I7	-	N
F2XM1	Compute 2x–1	N	N	F4	-	N
FABS	Absolute Value	N	N	F2	-	N
FADD/FADDP /FIADD	Add	N	N	F2	-	N
FBLD	Load Binary Coded Decimal	N	N	F1	-	N
FBSTP	Store BCD Integer and Pop	N	N	F1	-	N
FCHS	Change Sign	N	N	F2	-	N
FCLEX/FNCLEX	Clear Exceptions	N	N	F6	-	N
FCMOVcc	Floating-Point Conditional Move	N	N	F1	-	N
FCOM/FCOMP/FCOMPP	Compare Real	N	N	F3	-	N
FCOMI/FCOMIP/ FUCOMI/FUCOMIP	Compare Real and Set EFLAGS	N	N	F3	-	N
FCOS	Cosine	N	N	F4	-	N
FDECSTP	Decrement Stack-Top Pointer	N	N	F6	-	N
FDIV/FDIVP/FIDIV	Divide	N	N	F2	-	N
FDIVR/FDIVRP/FIDIVR	Reverse Divide	N	N	F2	-	N
FFREE	Free Floating-Point Register	N	N	F6	-	N
FICOM/FICOMP	Compare Integer	N	N	F3	-	N
FILD	Load Integer	N	N	F1	-	N
FINCSTP	Increment Stack-Top Pointer	N	N	F6	-	N
FINIT/FNINIT	Initialize Floating-Point Unit	N	N	F6	-	N
FIST/FISTP	Store Integer	N	N	F1	-	N
FLD	Load Real	N	N	F1	-	N

FLD1/FLDL2T /FLDL2E/FLD PI/FLDLG2/FL DLN2/FLDZ	Load Constant	N	N	F5	-	N
FLDCW	Load Control Word	N	N	F6	-	N
FLDENV	Load FPU Environment	N	N	F6	-	N
FMUL/FMUL P/FIMUL	Multiply	N	N	F2	-	N
FNOP	No Operation	N	N	F6	-	N
FPATAN	Partial Arctangent	N	N	F4	-	N
FPREM	Partial Remainder	N	N	F2	-	N
FPREM1	Partial Remainder	N	N	F2	-	N
FPTAN	Partial Tangent	N	N	F4	-	N
FRNDINT	Round to Integer	N	N	F2	-	N
FRSTOR	Restore FPU State	N	N	F6	-	N
FSAVE/FNSA VE	Store FPU State	N	N	F6	-	N
FSCALE	Scale	N	N	F2	-	N
FSIN	Sine	N	N	F4	-	N
FSINCOS	Sine and Cosine	N	N	F4	-	N
FSQRT	Square Root	N	N	F2	-	N
FST/FSTP	Store Real	N	N	F1	-	N
FSTCW/FNST CW	Store Control Word	N	N	F6	-	N
FSTENV/FNS TENV	Store FPU Environment	N	N	F6	-	N
FSTSW/FNST SW	Store Status Word	N	N	F6	-	N
FSUB/FSUBP/ FISUB	Subtract	N	N	F2	-	N
FSUBR/FSUB RP/FISUBR	Reverse Subtract	N	N	F2	-	N
FTST	TEST	N	N	F3	-	N
FUCOM/FUC OMP/FUCOM PP	Unordered Compare Real	N	N	F3	-	N
FXAM	Examine	N	N	F3	-	N
FXCH	Exchange Register Contents	N	N	F1	-	N
FXRSTOR	Restore FP and MMX™ State and Streaming SIMD Extension State	N	N	ST9	-	N
FXSAVE	Store FP and MMX™ State and Streaming SIMD Extension State	N	N	ST9	-	N
FXTRACT	Extract Exponent and Significand	N	N	F2	-	N
FYL2X	Compute y * log2x	N	N	F4	-	N
FYL2XP1	Compute y * log2(x +1)	N	N	F4	-	N
HLT	Halt	Y	Y	S	3A2	N
IDIV	Signed Divide	N	N	I2	-	N
IMUL	Signed Multiply	N	N	I2	-	N
IN	Input from Port	Y	Y	I1	3D	N

INC	Increment by 1	N	N	I2	-	N
INS/INSB/INSW/INSD	Input from Port to String	Y	Y	I8	3D	N
INT n/INTO/INT 3	Call to Interrupt Procedure	Y	N	I7	3C1	Y
INVD	Invalidate Internal Caches	Y	Y	S	3C2	N
INVLPG	Invalidate TLB Entry	Y	Y	S	3C2	N
IRET/IRETD	Interrupt Return	Y	N	I7	3C1	Y
Jcc	Jump if Condition Is Met	N	N	I7	-	N
JMP	Jump	Y	N	I7	3C1	Y
LAHF	Load Status Flags into AH Register	N	N	I9	-	N
LAR	Load Access Rights Byte	Y	N	S	3C1	Y
LDMXCSR	Load Streaming SIMD Extension Control/Status	N	N	ST9	-	N
LDS/LES/LFS/LGS/LSS	Load Far Pointer	Y	Y	I10	3C1	N
LEA	Load Effective Address	N	N	I11	-	N
LEAVE	High Level Procedure Exit	N	N	I7	-	N
LGDT/LIDT	Load Global/Interrupt Descriptor Table Register	Y	Y	S	3B1	N
LLDT	Load Local Descriptor Table Register	Y	Y	S	3B1	N
LMSW	Load Machine Status Word	Y	Y	S	3A2	N
LOCK	Assert LOCK# Signal Prefix	N	N	S	-	N
LODS/LODSB/LODSW/LODSD	Load String	N	N	I8	-	N
LOOP/LOOPcc	Loop According to ECX Counter	N	N	I7	-	N
LSL	Load Segment Limit	Y	N	S	3C1	Y
LTR	Load Task Register	Y	Y	S	3B1	N
MASKMOVQ	Byte Mask Write	N	N	ST8	-	N
MAXPS	Packed Single-FP Maximum	N	N	ST3	-	N
MAXSS	Scalar Single-FP Maximum	N	N	ST3	-	N
MINPS	Packed Single-FP Minimum	N	N	ST3	-	N
MINSS	Scalar Single-FP Minimum	N	N	ST3	-	N
MOV	Move	Y	N	I1	3C1	Y
MOV	Move to/from Control Registers	Y	Y	S	3B1	N
MOV	Move to/from Debug Registers	Y	Y	S	3B1	N
MOVAPS	Move Aligned Four Packed Single-FP	N	N	ST1	-	N
MOVD	Move 32 Bits	N	N	M1	-	N
MOVHLPS	High to Low Packed Single-FP	N	N	ST1	-	N
MOVHPS	Move High Packed Single-FP	N	N	ST1	-	N
MOVLHPS	Move Low to High Packed Single-FP	N	N	ST1	-	N
MOVLPS	Move Low Packed Single-FP	N	N	ST1	-	N
MOVMSKPS	Move Mask To Integer	N	N	ST1	-	N
MOVNTPS	Move Aligned Four Packed Single-FP Non Temporal	N	N	ST8	-	N
MOVNTQ	Move 64 Bits Non Temporal	N	N	ST8	-	N
MOVQ	Move 64 Bits	N	N	M1	-	N

MOVS/MOVSB/MOVSW/MOVSD	Move Data from String to String	N	N	I8	-	N
MOVSS	Move Scalar Single-FP	N	N	ST1	-	N
MOVSX	Move with Sign-Extension	N	N	I1	-	N
MOVUPS	Move Unaligned Four Packed Single-FP	N	N	F1	-	N
MOVZX	Move with Zero-Extend	N	N	I1	-	N
MUL	Unsigned Multiply	N	N	I2	-	N
MULPS	Packed Single-FP Multiply	N	N	ST3	-	N
MULSS	Scalar Single-FP Multiply	N	N	ST3	-	N
NEG	Two's Complement Negation	N	N	I2	-	N
NOP	No Operation	N	N	I11	-	N
NOT	One's Complement Negation	N	N	I4	-	N
OR	Logical Inclusive OR	N	N	I4	-	N
ORPS	Bit-wise Logical OR for Single-FP Data	N	N	ST5	-	N
OUT	Output to Port	Y	Y	I1	3D	N
OUTS/OUTSB/OUTSW/OUTSD	Output String to Port	Y	Y	I8	3D	N
PACKSSWB/PACKSSDW	Pack with Signed Saturation	N	N	M2	-	N
PACKUSWB	Pack with Unsigned Saturation	N	N	M2	-	N
PADDB/PADDW/PADDD	Packed Add	N	N	M3	-	N
PADDSB/PADDSW	Packed Add with Saturation	N	N	M3	-	N
PADDUSB/PADDUSW	Packed Add Unsigned with Saturation	N	N	M3	-	N
PAND	Logical AND	N	N	M4	-	N
PANDN	Logical AND NOT	N	N	M4	-	N
PAVGB/PAVGW	Packed Average	N	N	ST7	-	N
PCMPEQB/PCMPEQW/PCMPEQD	Packed Compare for Equal	N	N	M3	-	N
PCMPGTB/PCMPGTW/PCMPGTD	Packed Compare for Greater Than	N	N	M3	-	N
PEXTRW	Extract Word	N	N	ST7	-	N
PINSRW	Insert Word	N	N	ST7	-	N
PMADDWD	Packed Multiply and Add	N	N	M2	-	N
PMAXSW	Packed Signed Integer Word Maximum	N	N	ST7	-	N
PMAXUB	Packed Unsigned Integer Byte Maximum	N	N	ST7	-	N
PMINSW	Packed Signed Integer Word Minimum	N	N	ST7	-	N
PMINUB	Packed Unsigned Integer Byte Minimum	N	N	ST7	-	N
PMOVMSKB	Move Byte Mask To Integer	N	N	ST7	-	N
PMULHUW	Packed Multiply High Unsigned	N	N	ST7	-	N
PMULHW	Packed Multiply High	N	N	M2	-	N
PMULLW	Packed Multiply Low	N	N	M2	-	N

POP	Pop a Value from the Stack	Y	N	I1	3C1	Y
POPA/POPAD	Pop All General-Purpose Registers	N	N	I1	-	N
POPF/POPFD	Pop Stack into EFLAGS Register	Y	N	I9	3B1	Y
POR	Bitwise Logical OR	N	N	M4	-	N
PREFETCH	Prefetch	N	N	ST8	-	N
PSADBW	Packed Sum of Absolute Differences	N	N	ST7	-	N
PSHUFW	Packed Shuffle Word	N	N	ST7	-	N
PSLLW/PSLLD/PSLLQ	Packed Shift Left Logical	N	N	M5	-	N
PSRAW/PSRAD	Packed Shift Right Arithmetic	N	N	M5	-	N
PSRLW/PSRLD/PSRLQ	Packed Shift Right Logical	N	N	M5	-	N
PSUBB/PSUBW/PSUBD	Packed Subtract	N	N	M3	-	N
PSUBSB/PSUBSW	Packed Subtract with Saturation	N	N	M3	-	N
PSUBUSB/PSUBUSW	Packed Subtract Unsigned with Saturation	N	N	M3	-	N
PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ	Unpack High Packed Data	N	N	M2	-	N
PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ	Unpack Low Packed Data	N	N	M2	-	N
PUSH	Push Word or Doubleword Onto the Stack	Y	N	I1	3C1	Y
PUSHA/PUSHAD	Push All General-Purpose Registers	N	N	I1	-	N
PUSHF/PUSHFD	Push EFLAGS Register onto the Stack	Y	N	I9	3B1	Y
PXOR	Logical Exclusive OR	N	N	M4	-	N
RCL/RCR/ROL/ROR	Rotate	N	N	I5	-	N
RCPPS	Packed Single-FP Reciprocal	N	N	ST3	-	N
RCPSS	Scalar Single-FP Reciprocal	N	N	ST3	-	N
RDMSR	Read from Model Specific Register	Y	Y	S	3B1	N
RDPMC	Read Performance-Monitoring Counters	Y	Y	S	3B1	N
RDTSC	Read Time-Stamp Counter	Y	Y	S	3B1	N
REP/REPE/REPZ/REPNE/REPNZ	Repeat String Operation Prefix	N	N	I8	-	N
RET	Return from Procedure	Y	N	I7	3C1	Y
RSM	Resume from System Management Mode	Y	N	S	3A1	Y
RSQRTPS	Packed Single-FP Square Root Reciprocal	N	N	ST3	-	N
RSQRTSS	Scalar Single-FP Square Root Reciprocal	N	N	ST3	-	N

SAHF	Store AH into Flags	N	N	I9	-	N
SAL/SAR/SHL /SHR	Shift	N	N	I5	-	N
SBB	Integer Subtraction with Borrow	N	N	I2	-	N
SCAS/SCASB/ SCASW/SCAS D	Scan String	N	N	I8	-	N
SETcc	Set Byte on Condition	N	N	I6	-	N
SFENCE	Store Fence	N	N	ST8	-	N
SGDT/SIDT	Store Global/Interrupt Descriptor Table Register	Y	N	S	3B1	Y
SHLD	Double Precision Shift Left	N	N	I5	-	N
SHRD	Double Precision Shift Right	N	N	I5	-	N
SHUFPS	Shuffle Single-FP	N	N	ST6	-	N
SLDT	Store Local Descriptor Table Register	Y	N	S	3B1	Y
SMSW	Store Machine Status Word	Y	N	S	3B1	Y
SQRTPS	Packed Single-FP Square Root	N	N	ST3	-	N
SQRTSS	Scalar Single-FP Square Root	N	N	ST3	-	N
STC	Set Carry Flag	N	N	I9	-	N
STD	Set Direction Flag	N	N	I9	-	N
STI	Set Interrupt Flag	Y	Y	I9	3A2	N
STMXCSR	Store Streaming SIMD Extension Control/Status	N	N	ST9	-	N
STOS/STOSB/ STOSW/STOS D	Store String	N	N	I8	-	N
STR	Store Task Register	Y	N	S	3C1	Y
SUB	Subtract	N	N	I2	-	N
SUBPS	Packed Single-FP Subtract	N	N	ST3	-	N
SUBSS	Scalar Single-FP Subtract	N	N	ST3	-	N
SYSENTER	Fast Transition to System Call Entry Point	Y	N	S	3C1	Y
SYSEXIT	Fast Transition from System Call Entry Point	Y	Y	S	3C1	N
TEST	Logical Compare	N	N	I6	-	N
UCOMISS	Unordered Scalar Single-FP compare and set EFLAGS	N	N	I11, ST4	N	N
UD2	Undefined Instruction	N	N	-	-	N
UNPCKHPS	Unpack High Packed Single-FP Data	N	N	ST6	-	N
UNPCKLPS	Unpack Low Packed Single-FP Data	N	N	ST6	-	N
VERR/VERW	Verify a Segment for Reading or Writing	Y	N	S	3C1	Y
WAIT/FWAIT	Wait	N	N	F6	-	N
WBINVD	Write Back and Invalidate Cache	Y	Y	S	3C2	N
WRMSR	Write to Model Specific Register	Y	Y	S	3B1	N
XADD	Exchange and Add	N	N	I1	-	N
XCHG	Exchange Register/Memory with Register	N	N	I1	-	N
XLAT/XLATB	Table Look-up Translation	N	N	I11	-	N
XOR	Logical Exclusive OR	N	N	I4	-	N
XORPS	Bit-wise Logical Xor for Single-FP Data	N	N	ST5	-	N

LIST OF REFERENCES

1. Goldberg, Robert, *Architectural Principles for Virtual Computer Systems*, Ph.D. Dissertation, Harvard University, Cambridge, Massachusetts, October 1972.

2. Bugnion, Edouard, Devine, Scott, Govil, Kinshuk, and Rosenblum, Mendel, "Disco: Running Commodity Operating Systems on Scalable Multiprocessors," *ACM Transactions on Computer Systems, v.* 15.4, pp. 412-447, November 1997.

3. Popek, Gerald J. "Formal Requirements for Virtualizable Third Generation Architectures," *Communications of the ACM, v* 17.7, pp. 412-421, July 1974.

4. National Security Agency Report CSC-EPL-92/003, *Final Evaluation Report: HFS Incorporated, XTS-200*, 27 May 1992.

5. National Security Agency Report NCSC-FER-94/34, *Final Evaluation Report: Gemini Computer, Inc Gemini Trusted Network Processor Version 1.01*, 28 June 1995.

6. Intel Corporation, *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference*, 1999.

7. VMware, Inc, "Welcome to VMware, Inc. – Virtual Platform Technology." [http://www.vmware.com]. March 1999.

8. Karger, Paul A., Zurko, Mary Ellen, Bonin, Douglas W., Mason Andrew H., Kahn, Clifford E., "A VMM Security Kernel for the VAX Architecture," *IEEE*, pp. 2-19, 1990.

9. Department of Defense, "*Department of Defense Trusted Computer System Evaluation Criteria," DOD 5200.28-STD, December 1985.*

10. Hall, Judith., and Robinson, Paul T., "Virtualizing the VAX Architecture," *Proceedings of the 18th International Symposium on Computer Architecture*, pp.380-389, Toronto, Canada, May 1991.

11. Nutt, Gary, *Operating systems: A Modern Perspective*, pp. 255-291, Addison-Wesley, 1997.

12. Digital Equipment Corporation, Order No. AA-LA39A-TE, *VMS Analyze/Disk_Structure Utility Manual*, April 1988.

13. National Security Agency Report CSC-EPL-91/005, *Final Evaluation Report: Boeing Space and Defense Group, MLS LAN Secure Network Server System*, 28 August 1991.

14. National Security Agency Report CSC-EPL-90/001.A, *Final Evaluation Report: Verdix Corporation VSLAN 5.1/VSLANE 5.1*, 11 January 1994.

15. National Security Agency Report CSC-EPL-92/001.A, *Final Evaluation Report: Trusted Information Systems, Inc. Trusted XENIX Version 4.0*, January 1994.

16. Schroeder, Michael D. and Saltzer, Jerome H., "A Hardware Architecture for Implementing Protection Rings," *Communications of the ACM*, v. 15.3, pp.157-169, March 1972.

17. Anderson, James P., *Computer Security Technology Planning Study, Vol. 1*, Hanscom AFB ESD-TR-73-51, 1972.

18. Sibert, Olin, Porras, Phillip A., Lindell, Robert, "The Intel 80x86 Processor Architecture: Pitfalls for Secure Systems," *Institute of Electrical and Electronics Engineers*, 1995.

19. Lampson, Butler W., "A Note on the Confinement Problem," *Communications of the ACM*, v. 16.10, pp.613-615, October 1973.

20. Lawton, Kevin, "Running Multiple Operating Systems Concurrently on an IA32 PC Using Virtualization Techniques." [http://www.freemware.org/docs.phtml?file=paper.txt]. June 1999.

21. Lawton, Kevin, "Welcome to the 'Bochs Software Company' Home Page." [http://www.bochs.com]. July 1999.

22. Wine, "Wine Development HQ." [http://www.winehq.com]. July 1999.

23. *Common Criteria for Information Technology Security Evaluation, Version 2.0*, CCIB-98-026, May 1998

24. Dr. Dobbs' Journal, "Intel Secrets, Bugs, and Undocumented Op Codes." [http://www.x86.org/secrets/]. July 1999.

25. Rosenblum, Mendel, VMware, Inc., Lecture at Stanford University, Palo Alto, California, August 17, 1999.

26. Intel Corporation, *Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture*, 1999.

27. Intel Corporation, *Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide*, 1999.

BIBLIOGRAPHY

1. Andover.Net, "SlashDot: News for Nerds, Stuff That Matters." [http://www.slashdot.org]. May 1999.

2. Lecture by Steven B. Lipner, Mitretek Systems, at the Naval Postgraduate School, Monterey, CA, 25 February 1999.

3. Trusted Information Systems, A Proposed Interpretation of the TCSEC for Virtual Machine Monitors, Vol 1: Strict Separation, 1 May 1990.

INITIAL DISTRIBUTION LIST

No. Copies

1. Defense Technical Information Center 2
   8725 John J. Kingman Rd., STE 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library 2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, CA 93943-5101

3. Chairman, Code CS 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943-5000

4. Dr. Cynthia E. Irvine 2
   Computer Science Department, Code CS/Ic
   Naval Postgraduate School
   Monterey, CA 93943

5. Steven B. Lipner 1
   Mitretek Systems
   7525 Colshire Drive
   McLean, VA 22102-7400

6. Paul Pittelli 1
   National Security Agency
   Research and Development Building
   R2
   9800 Savage Road
   Fort Meade, MD 20755-6000

7. CAPT Dan Galik 1
   Space and Naval Warfare Systems Command
   PMW 161
   Building OT-1, Room 1024
   4301 Pacific Highway
   San Diego, CA 92110-3127

8. Commander, Naval Security Group Command 1
Naval Security Group Headquarters
9800 Savage Road
Suite 6585
Fort Meade, MD 20755-6585

9. Mr. George Bieber 1
Defense Information Systems Agency
Center for Information Systems Security
5113 Leesburg Pike, Suite 400
Falls Church, VA 22041-3230

10. Mr. Jim Throneberry 1
N643
Presidential Tower 1
2511 South Jefferson Davis Highway
Arlington, VA 22202

11. Mr. John Mildner 1
Director of Technical Operations
Code 72A
SPAWAR Systems Center Charleston
P.O. Box 190022
North Charleston, SC 29419

12. John Scott Robin 1
9129 Sharee Place
Denham Springs, LA 70726